High Resolution Simulation of Synthetic Aperture Radar Imaging

by

Cindy Romero

A Thesis presented to the Faculty of the

California Polytechnic State University, San Luis Obispo

In partial fulfillment

of the requirements for the degree,

Master of Science in Electrical Engineering

June 2010

Supported by Raytheon Space and Airborne Systems Division

COMMITTEE MEMBERSHIP

TITLE: High Resolution Simulation of Synthetic Aperture Radar Imaging

AUTHOR: Cindy Romero

DATE SUBMITTED: June 2010

COMMITTEE CHAIR: Professor John Saghri

COMMITTEE MEMBER: Professor Xiao-Hua Yu

COMMITTEE MEMBER: Professor Wayne Pilkington

ABSTRACT

High Resolution Simulation of Synthetic Aperture Radar Imaging

by Cindy Romero

The goal of this Master's thesis is to develop a more realistic simulation of Synthetic Aperture Radar (SAR) that has the ability to image detailed targets, and that can be used for Automatic Target Recognition (ATR). This thesis project is part of ongoing SAR ATR research at California Polytechnic State University (Cal Poly) sponsored by Raytheon Space & Airborne Systems and supervised by Dr. John Saghri.

SAR is a form of radar that takes advantage of the forward motion of an antenna mounted on a moving platform (such as an airplane or spacecraft) to synthetically produce the effect of a longer antenna. Since most SAR images used for military ATR are classified and not available to the general public, all academic research to date on ATR has been limited to a small data set of Moving and Stationary Target Acquisition and Recognition Radar (MSTAR) images. Due to the unavailability of radar equipment or a greater range of SAR data, it has been necessary to create a SAR image generation scheme in which the parameters of the radar platform can be directly modified and controlled to be used for ATR applications.

This thesis project focuses on making several improvements to Matthew Schlutz's 'Synthetic Aperture Radar Imaging Simulated in Matlab' thesis. First, the simulation is optimized by porting the antenna pattern and echo generator from Matlab to C++, and the efficiency of the code is improved to reduced processing time. A three-dimensional (3-D) graphics application called Blender is used to create and position the target models in the scene imaged by the radar platform and to give altitude, target range (range of closest approach from the platform to the

center area of the target) and elevation angle information to the radar platform. Blender allows the user to take pictures of the target as seen from the radar platform, and outputs range information from the radar platform plane to each point in the image. One of the major advantages of using Blender is that it also outputs range and reflectivity information about each pixel in the image. This is a significant characteristic that was hardcoded in the previous theses, making those simulations less realistic.

For this thesis project, once the target scene is created in Blender, an image is rendered and saved as an OpenEXR file. The image is rendered in orthographic mode, which is a form of projection whereby the target plane is parallel with the projection plane. This parameter means that the simulation cannot image point targets that appear and disappear during the platform motion. The echo generation program then uses the range and reflectivity obtained from the OpenEXR file, the optimized antenna pattern, and several other user defined parameters to create the echo (received signal). Once the echo is created in the echo generation program, it is then read into Matlab in order for it to go through the Range Doppler Algorithm (RDA) and then output the final SAR image.

ACKNOWLEDGMENTS

I would firstly like to thank my thesis advisor, Dr. John Saghri, for giving me the opportunity to be part of his research team and for the guidance and support he gave me throughout this project. I would also like to thank Raytheon Space and Airborne Systems, particularly Jeff Hoffner, for their continued support and sponsorship of this research project. For their previous research on SAR simulations undertaken at Cal Poly, I would like to acknowledge Lynn Kendrick, Brian Zaharris, Paul Mason and Matthew Schlutz.

I would like to express my gratitude to my coworker, colleague, advisor and good friend, Matthew Kuiken, for all of his generous support and advice. I wouldn't have been able to complete this project without Matt's help, and I will never be able to repay him for all the time and support he gave me throughout this project.

I would also like to thank Dr. Wayne Pilkington and Dr. Helen Yu for their participation, time, and advice as members of the thesis committee.

Special thanks to my boyfriend and best friend, Sam McLeod, for keeping me motivated throughout this project and for helping me in the editing of the paper.

Finally, I would like to thank my family and friends for their continuous support and encouragement during my time at Cal Poly.

TABLE OF CONTENTS

LIST OF FIGURES

1. SYNTHETIC APERTURE RADAR

Synthetic Aperture Radar (SAR) is a form of radar that processes multiple radar images to produce a higher-resolution image. The multiple radar images are obtained by mounting an antenna to a moving platform, such as an airplane or satellite, and illuminating the target scene at different distances and times. The antenna produces a series of beams that widen as they make their way towards a long-distance reflecting target, as shown in Figure 1 (below). The echo waveforms received at the different antenna positions are then post-processed to obtain useful information about the size and shape of the reflecting target. [1,3,5]

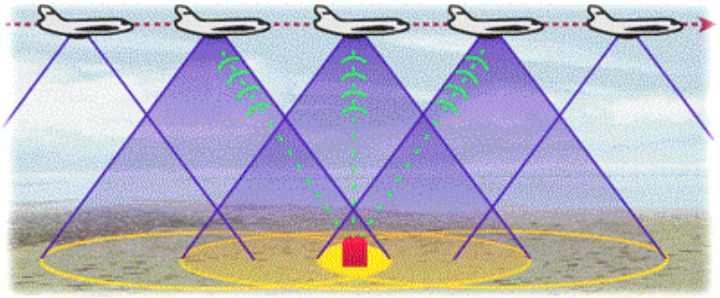

Figure 1: Synthetic Aperture Radar[10]

A large antenna producing a narrow beam on the target is needed in order to maximize the resolution of the radar images, thus making it easier to distinguish target features. The wider the beam on the target, the more difficult it is to discern which reflections are coming from the target and thus to locate the target. Many beams can be sent from and received by the moving antenna platform with respect to a single target. The reflections received by the antenna platform are analyzed together by using the Range Doppler Algorithm (RDA) in order to produce an accurate and relatively high resolution two-dimensional (2-D) image. [1,4,5]

1.1. Brief history of SAR

Radar was initially developed for use by the military during World War II, with the main purpose of tracking aircraft and ships. The advantage of the radio frequency used was that it could penetrate through heavy weather and darkness to locate targets that would not have been detected by using remote sensing with visible light. [1,5]

Radar systems can measure; the range to the target from the time it takes the radar pulse to travel to and from the target, the direction of the target via antenna directivity, and the target speed by using the Doppler shifts. In June 1951, Carl A. Wiley, a mathematician at Goodyear Aircraft Corporation in Litchfield Park, discovered that finer resolution could be obtained by processing the Doppler shifts. Using the Doppler shift processing technique and wavefront reconstruction theory, Wiley developed Synthetic Aperture Radar (SAR) by which 2-D images of targets and the earth's surface could be constructed using radar. The method was termed Synthetic Aperture Radar because its signal analysis created the effect of a very long antenna. [1,5]

In the 1950s and 1960s, remote sensing technology was developed further, but it wasn't until the 1970s that military SAR technology was released to the civilian community. SAR images were found to provide a complimentary and useful addition to optical sensors. Early work in SAR technology was done on aircraft, but in June 1978, NASA launched its first earth orbiting satellite designed for remote sensing of the earth's oceans, 'SEASAT' (illustrated in Figure 2 below). The parameters used for the SEASAT mission were; a satellite altitude of 800 kilometers, a radar frequency of 1.275GHz (L-band), a swath width of 100 km, an angle of incidence of 23 degrees, and a resolution of 25 meters in the range and azimuth directions. The

range is the direction perpendicular to the movement of the platform and the azimuth is the direction parallel to the movement of the platform. [1,8]

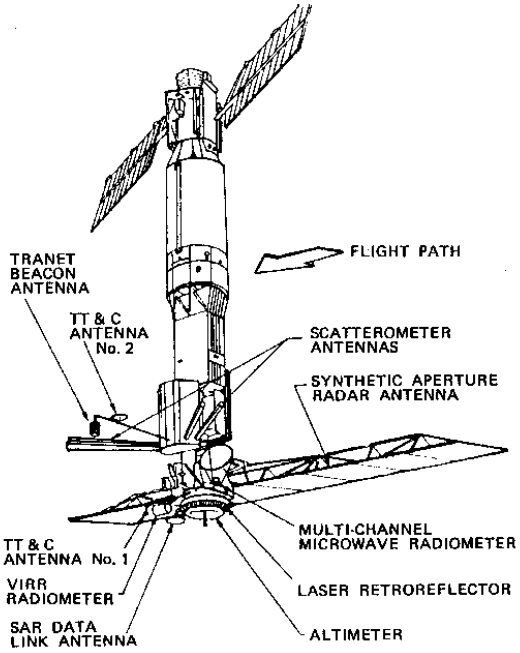

Figure 2: SEASAT[11]

The SEASAT's radar system collected SAR data and recorded it on black and white film. However, the data collected appeared unfocused and needed to be sent through a phase-sensitive processor to obtain a focused image. By using Fourier Optics principles, it was found that data could be focused by using laser beams and lenses. This technique was then implemented digitally with the use of Fast Fourier Transforms (FFT) instead of lenses. [1,4]

The SEASAT mission prompted research to develop digital SAR processors for satellites. The first digital SAR processor was built for SEASAT in 1978 and took 40 hours to process a 40 by 40 kilometer image with 25-meter resolution. Today, average computers can process the same image in less than a quarter of a second. The digital processor, when implemented on SEASAT, proved to be operational and effective, and since then has been the standard method used. [1,4]

1.2. History of SAR simulation research at Cal Poly

Since 2005, Cal Poly research students have incrementally developed code which processes raw SAR signals to generate images of targets, with the ultimate aim of being able to use these images for Automatic Target Recognition (ATR). Lynn Kendrick began the development of the SAR simulation as her Cal Poly Senior project in 2005 and was successfully able to model a stationary point target in one dimension. Kendricks' simulation consisted of only range and no moving platform. This meant that a larger antenna could not be synthetically obtained and there was no range cell migration or azimuth processing phase. In order to find the range of point targets in the single range direction, Kendricks coded and simulated the basic filtering mechanism of the RDA.[12]

It wasn't until 2007 that a full 2-D SAR simulation and RDA was developed as a Master's thesis by Brian Zaharris. Zaharris created a Matlab simulation that arrayed stationary and slowly moving point targets in a 2-D plane of azimuth and range. Zaharris' simulation tracked slowly moving targets by applying Kalman filtering on the raw SAR signal portion of the code to obtain accurate images of the targets. Paul Mason, a Master's student at Cal Poly who also worked on his project in 2007, used the range-Doppler algorithm that Zaharris had created and adapted it to 2-D geometrical shapes and letters composed of arrays of point targets, with the azimuth and range location of each point defined in input profiles. Masons' 2-D simulation took a longer amount of time to simulate than Zaharris' and therefore Mason used Zaharris' 2-D simulation as a starting point to implement 3-D SAR imaging. Due to time constraints, Mason decided to focus on the 2-D SAR simulation. At that time, several limitations needed to be addressed in the 2-D SAR simulation before trying to move into 3-D SAR simulations.

Each point target in the 2-D SAR simulation required a reflection to be calculated and, due to the complexity of the radar reflection equation used, it would take 30 seconds to calculate the reflection for each point target. A simulation of MSTAR images, which are 128 by 128 pixels, would take over five days to complete. The memory-intensive part of the code came from saving the reflections from each point target separately. A lack of memory allocation also limited Zaharris' simulation to 25 point targets. [6, 9]

In Fall 2007, Matthew Schlutz got involved on this ongoing project as part of his Master's thesis. Schlutz focused on improving the SAR simulation for application to more complex 2-D targets and simple 3-D targets. First, he made modifications to the simulation to sum all 256 reflections and reduce the raw SAR signal space to a single 2-D double array. He then rewrote the echo generation section of the code to optimize processing efficiency. It would have taken 10 minutes to generate the SAR echoes from 11 point targets using Zaharris' code, and therefore the processing time for a 16 by 16 pixel image would have been over four hours. With Schlutz' modifications, a 16 by 16 pixel image would take 10 minutes to process a SAR image. Schlutz added a new target import feature to the SAR simulation, which he used to specify the location and reflectivity of point targets based on the location and intensity of the pixels in the imported image. Schlutz also created 3-D simulations by importing multiple 2-D azimuth/range profiles at different altitudes.[5, 9]

Although Schlutz made useful improvements to the simulation, there were still a lot of drawbacks that needed to be addressed in order to develop a simulation that could be used for ATR. The SAR echo generation sequence needed to be optimized to allow larger images to be generated at higher resolution. The reflecting point targets had been hardcoded and therefore image obscurities and other difficulties with ATR would be hard to model. The focus of this

thesis is to address these drawbacks by looking at ways in which the simulation can be optimized.

1.3 SAR geometry and dimensions

There are two directions that need to be taken into account when considering a moving radar platform and a point target (as shown in Figure 3 below). The azimuth direction is defined as the direction aligned with the platform velocity vector, in this case the aircraft. The range direction is the direction in which the transmitted pulse travels, and is perpendicular to the azimuth direction. The altitude direction is then omitted in a 2-D simulation. As the radar platform moves, a beam consisting of hundreds of transmitted radar pulses illuminates an area on the ground. This area is referred to as the footprint, and its size is determined by the antenna pattern, the beamwidth, and the range distance to the ground.[2, 9]

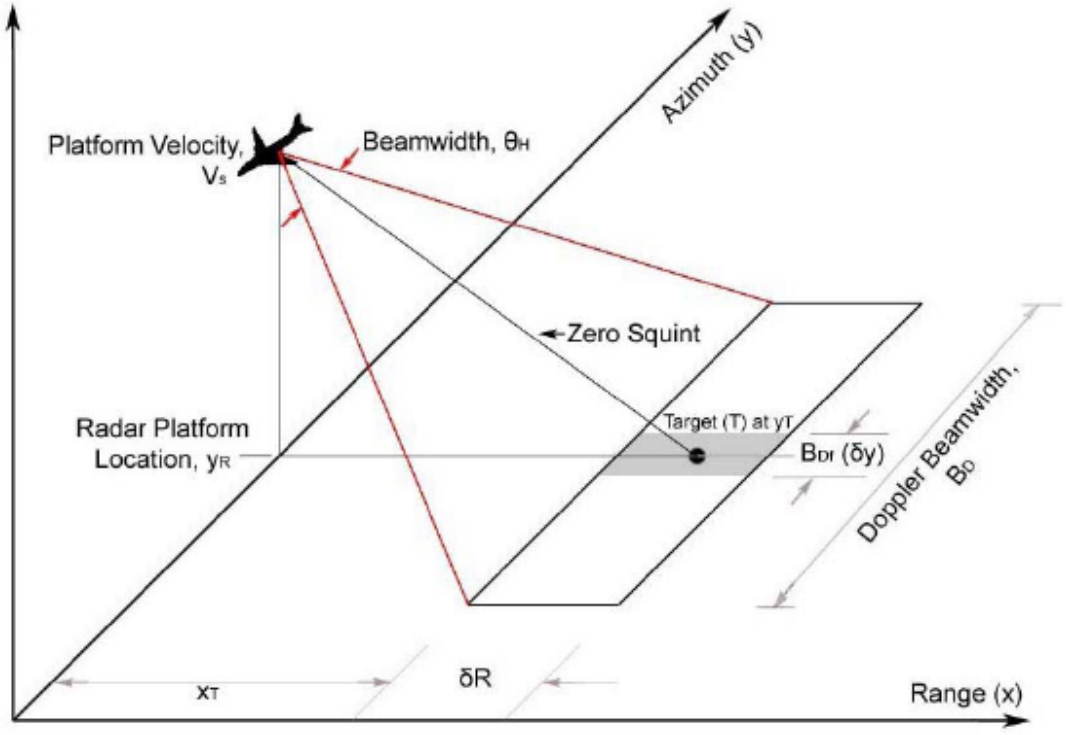

Figure 3: SAR geometry and dimensions

1.4 Modes of SAR operation

A SAR can be operated in various modes but only three of them are in practice today; the stripmap, scan, and spotlight SAR modes (as illustrated in Figure 4 below). In stripmap SAR mode, as the radar platform moves, the ground swath is illuminated with a continuous sequence of pulses while the antenna beam is held constant in elevation and azimuth. The result is an image strip with continuous image quality in the azimuth direction. The scan SAR mode is a variation of the stripmap mode, which provides large area coverage by having the antenna scan in range several times along the flight path. The azimuth resolution of a scan SAR product is lower than in stripmap SAR mode because of its reduced azimuth bandwidth. Spotlight SAR mode is obtained by increasing the angle of the illumination on the desired location and staying fixed on a target as the platform moves. The spotlight mode is mainly used when the location of the target of interest is previously known. This thesis uses stripmap mode for all simulations. [2,6]

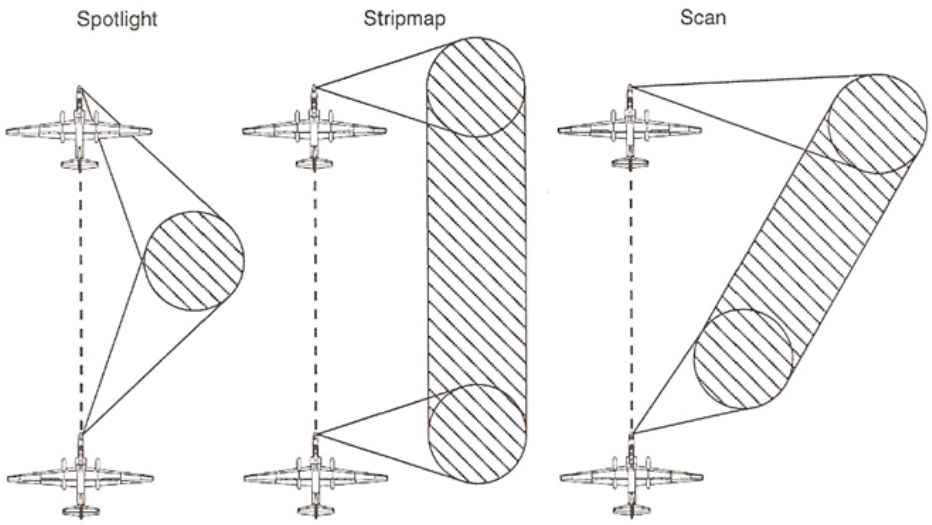

Figure 4: Three modes of SAR operation[10]

1.5 Transmitted pulse

The radar sends out an FM pulse in the range direction, which is described as follows:

$$s_{pul}(t) = w_r(t)\cos\left(2\pi f_c t + \pi K_r t^2\right) \quad (1)$$

In Equation 1: the signal pulse, s_{pul}, is a function of range time or quick time, t; $w_r(t)$ is a rectangular function that represents the pulse duration as a function of quick time; f_c is the carrier frequency; and K_r is the chirp rate. When the chirp rate sign is positive, the pulse is referred to as an 'up chirp' because the pulse frequency increases with time. In the same way, a negative chirp rate is referred to as a 'down chirp'. Both up chirp and down chirp achieve the same result and therefore the choice of the sign is up to the designer.[2]

Other important parameters are the signal bandwidth, *Bo*, and the chirp pulse duration, T_r, (both defined in Equation 2); and the range resolution, ρ_r, (defined in Equation 3).

$$Bo = |K_r| t_r \quad (2)$$

$$\rho_r = \frac{c}{2B_o} \quad (3)$$

The transmitted radar pulses, as shown in Figure 5 (below), are evenly spaced and repeated at a precisely controlled time interval, called the Pulse Repetition Interval (PRI). The inverse of this interval is the Pulse Repetition Frequency (PRF). Figure 5 shows the transmitted radar signal as cosine with a linearly ramping-up frequency over a transmit duration, followed by a receive duration. [2]

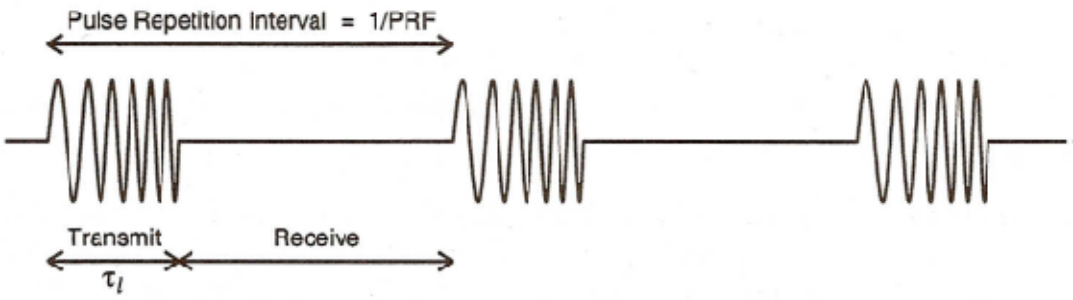

Figure 5: Transmitted radar pulse[2]

Figure 6 illustrates a timeline of the radar signal magnitude at the antenna during the transmitted pulses and received echoes. The raw SAR signal space is the result of plotting the magnitude of each range slice echo as a function of range and azimuth. [2]

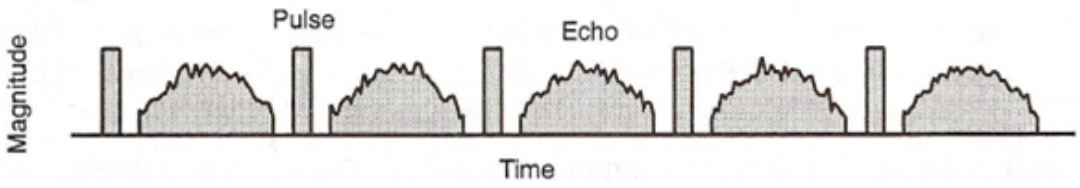

Figure 6: Transmit and receive cycles of a pulsed radar[2]

1.6. Received signal

Once the raw SAR signal is received, it goes through a quadrature demodulation process. Quadrature demodulation is the process of taking the signal received at the antenna, demodulating it to remove the high frequency carrier, and then sampling the demodulated data in discrete time. Quadrature demodulation causes the signal to be complex and have phase and magnitude terms. Equation 4 represents the raw SAR received radar signal, $s_r(t, \eta)$, after quadrature demodulation. Equation 4 shows that the return signal is directly proportional to the reflectivity of the target and to the time duration after the pulse is transmitted. [2, 5]

$$s_{rx}(t,\eta) = \sum_{m=0}^{M-1}\left[F_n w_r\left(t-\frac{2R_m(\eta)}{c}\right) w_a(\eta-\eta_c) e^{-j4\pi\left(\frac{f_o R_m(\eta)}{c}\right)+j\pi K_r\left(t-\frac{2R_m(\eta)}{c}\right)^2}\right] + n_m(t,\eta) \quad (4)$$

The received signal is a function of two time variables; range time or quick time, t; , and azimuth time (or slow time), η. F_n is the reflectivity of the target (a constant between zero and one), $\frac{2R_m(\eta)}{c}$ is the time delay, $w_a(\eta-\eta_c)$ is the azimuth beam pattern amplitude modification, and $n_m(t,\eta)$ is the Additive White Gaussian Noise (AWGN). η, is the azimuth time, and η_c is the azimuth time at which the center of the beam pattern crosses the center target area, also called the time of zero Doppler. [2]

The time delay is calculated as twice the instantaneous slant range divided by the speed of light. To be able to define the instantaneous slant range to the target, it is assumed that the aircraft is flying at a uniform velocity in a straight path in accordance with the azimuth direction, and the curvature of the Earth is neglected. The range equation, $R_m(\eta)$, is defined in Equation 5.

$$R_m(\eta) = \sqrt{{R_{o_m}}^2 + {V_r}^2\,\eta^2} = \sqrt{(X_o+x_m)^2 + {V_p}^2\left(\eta+\frac{y_m}{V_p}\right)^2} \quad (5)$$

Since most SAR antennas are unweighted in the azimuth plane, the one-way beam pattern can be approximated with a sinc function as shown in Equation 6.

$$\rho_a(\theta) \approx sinc\left(\frac{0.886\,\theta}{\beta_{bw}}\right) \quad (6)$$

The above equation uses θ to represent the angle measured from boresight in the slant range plane and β_{bw} to represent the azimuth beam width. The azimuth beam width is calculated in Equation 7 and is inversely proportional to the aperture antenna length,L_a, in the azimuth direction. The aperture is the 'opening' through which the sensor views the imaged target. The

azimuth beam pattern is given by the square of $\rho_a(\theta)$, because of the two-way propagation of the radar energy, and is expressed a function of azimuth time, η , as shown in Equation 8. [2]

$$\beta_{bw} = 0.886\lambda/L_a \tag{7}$$

$$\omega_a(\eta) \approx \rho_a^2\{\theta(\eta)\} \approx sinc^2\left(\frac{0.886\theta(\eta)}{\beta_{bw}}\right) \tag{8}$$

The azimuth beam pattern amplitude modification, $w_a(\eta - \eta_c)$, and its effect on signal strength and Doppler frequency is shown in Figure 7 (below). The top part of Figure 7 shows how a target on the ground is illuminated by several pulses as the platform moves. The azimuth beam pattern causes the strength of the signal to vary for each pulse. The top part of Figure 7 also shows the azimuth beam pattern for a zero squint. The target is just entering the main lobe of the beam at position A. The middle part of Figure 7 shows how the received signal strength is governed by the azimuth beam pattern. The received signal strength increases until the target lies in the center of the beam, as shown in position B, and then decreases until the target lies in the first null of the beam pattern, as shown in position C. The largest reflection strength is produced by the center node of the beam pattern, but the side nodes of the beam pattern also produce small amounts of energy. The bottom part of Figure 7 shows the Doppler frequency history of the target, which is proportional to the target's radial velocity with respect to the sensor. The Doppler frequency is positive when the target is approaching the radar and it is negative when the target is receding. [2]

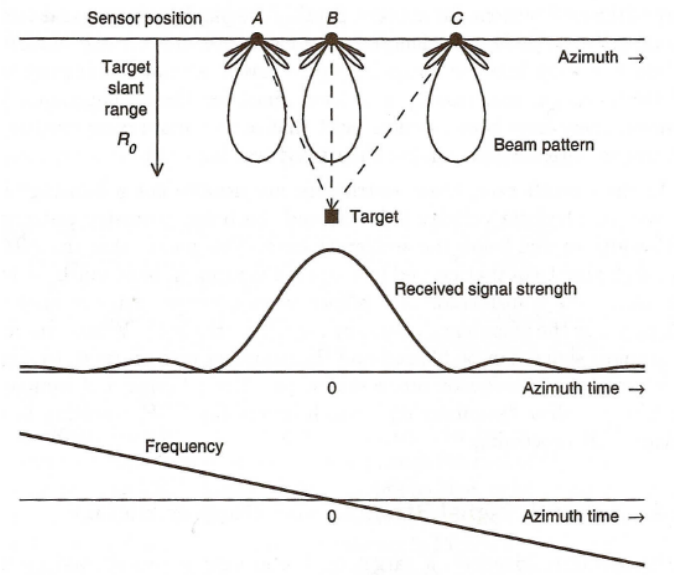

Figure 7: Doppler frequency history of the target[2]

An important parameter obtained from a SAR system is the azimuth resolution, ρ_a, as shown in Equation 9, The azimuth resolution is a function of: the radar beam ground velocity, V_g; the squint angle, θ_{sq}; and the Doppler bandwidth, Δf_{dop}. The calculation of the Doppler bandwidth is shown in Equation 10. However, for simplification between the airplane platform case and the satellite case, the azimuth resolution is approximately one-half of the antenna length and independent of range, velocity or wavelength. [2]

$$\rho_a = \frac{0.886 V_g cos\theta_{sq,c}}{\Delta f_{dop}} \approx \frac{L_a}{2} \tag{9}$$

$$\Delta f_{dop} = \frac{2V_s cos\theta_{sq}}{\lambda} \theta_{bw} \tag{10}$$

The azimuth resolution without SAR processing, ρ'_a, shown in Equation 11, is the projection of the beamwidth onto the ground and is called the real aperture radar resolution.

$$\rho'_a = R(n_c)\theta_{bw} = \frac{0.886 R(n_c)\lambda}{L_a} \tag{11}$$

The squint angle, θ_{sq}, labeled in Figure 8 (below) and derived in Equation 12, is the angle between the slant range vector and the zero Doppler plane. The squint angle varies as a function of azimuth time and therefore decreases as the platform approaches the target and increases as the platform moves away from the target. The calculation of the maximum squint angle, $\theta_{sq_{max}}$, is shown in Equation 13. [2]

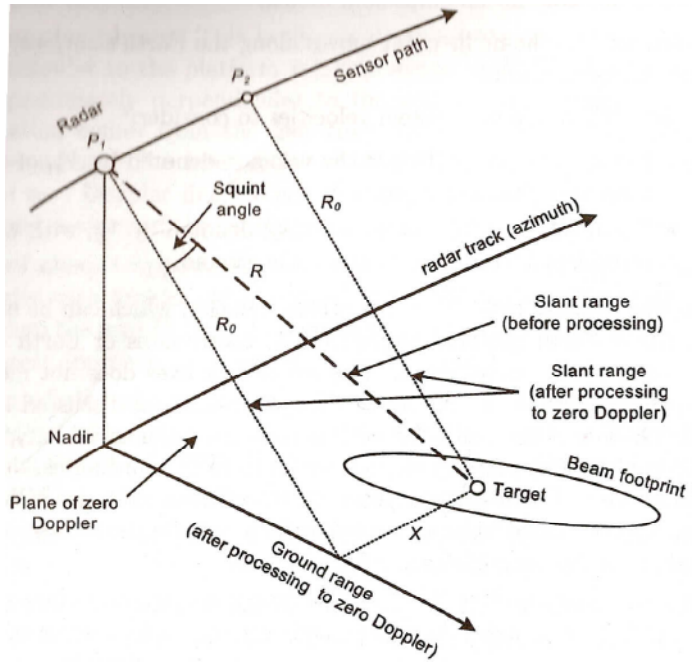

Figure 8: SAR slant range and squint angle geometry[2]

$$\theta_{sq} = arccos\left(\frac{R_{o_m}}{R_m(\eta)}\right) \tag{12}$$

$$\theta_{sq_{max}} = arccos\left(\frac{R_{o_m}}{R_m(\eta)}\right)\Big|_{R_m(\eta)=\left(\frac{dur}{2}\right)V_p} = arccos\left(\frac{2R_{o_m}}{durV_p}\right) \tag{13}$$

Figure 9 below shows how the received data is written into the 2-D signal processor memory.

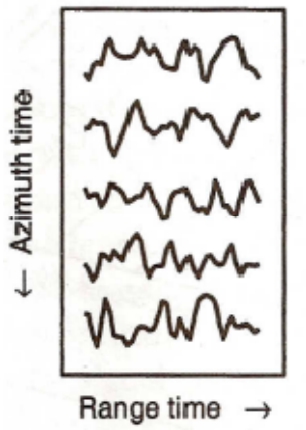

Figure 9: How the signal data is written into memory[2]

2. RANGE DOPPLER ALGORITHM

The Range Doppler Algorithm (RDA) is designed to process raw SAR data, using frequency domains and matched filtering to generate an image of a point target. The RDA performs matched filtering (see section 2.1 below) separately in the Fast Fourier Transformed (FFT) range and azimuth domain. The RDA uses Range Cell Migration Correction (RCMC) to separate the processing of range and azimuth data. RCMC (see section 2.2 below) is performed in the range and azimuth frequency domain, which is referred to as the range Doppler domain.[5]

The block diagram in Figure 10 shows the processing steps involved in the RDA.

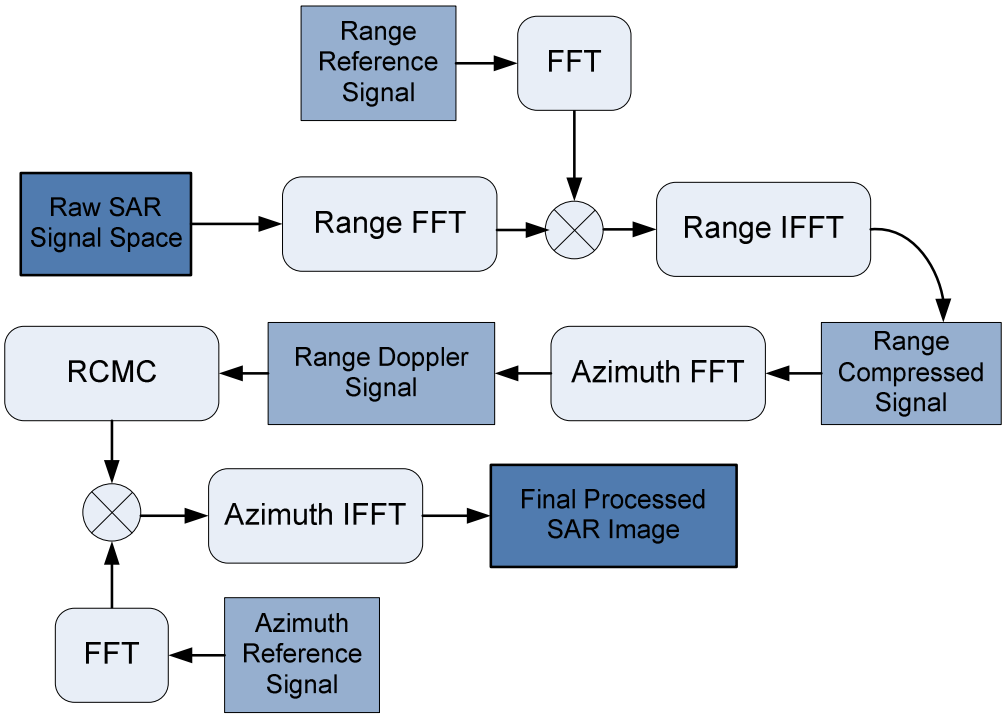

Figure 10: Range Doppler Algorithm block diagram[5]

The raw SAR 2-D signal input is first analyzed as a series of range time signals for each azimuth bin. The next step is to obtain range compression by taking the Fourier transform of the raw data

with respect to the range and multiplying it by the complex conjugate of the Fourier transform of the reference signal. Each signal is then transformed back into the range time domain by taking the range Inverse Fast Fourier Transform (IFFT). The result is the range compressed signal.[5, 9]

The range compressed signal is then composed into a series of signals with respect to azimuth time at different range bins. An azimuth FFT is then applied to each azimuth signal and RCMC is performed in the range Doppler domain before azimuth matched filtering. Azimuth matched filtering of each signal is then performed and transformed back into the time domain by taking the azimuth inverse fast Fourier transforms (IFFTs). The result is the final target image. [5, 9]

2.1. Matched filtering

As stated above, the RDA uses matched filtering to process raw data into images. Matched filtering is the correlation of a reference signal with an unknown signal, and is equivalent to the convolution of the unknown signal with the conjugated time-reversed reference signal. The matched filtering process requires a reference signal that is ideal, noiseless and centered in the middle of the footprint. [1, 5]

Figure 11 below shows a matched filter example where the transmitted radar signal is *s(t)*, the received radar signal is modeled as a time delayed version of *s(t)* and the time reversed version of s(t) is the matched filter template, *h(t).* The convolution of the received signal and the matched filter produces a compressed pulse of energy centered around the time delay of radar reflection.[1,5]

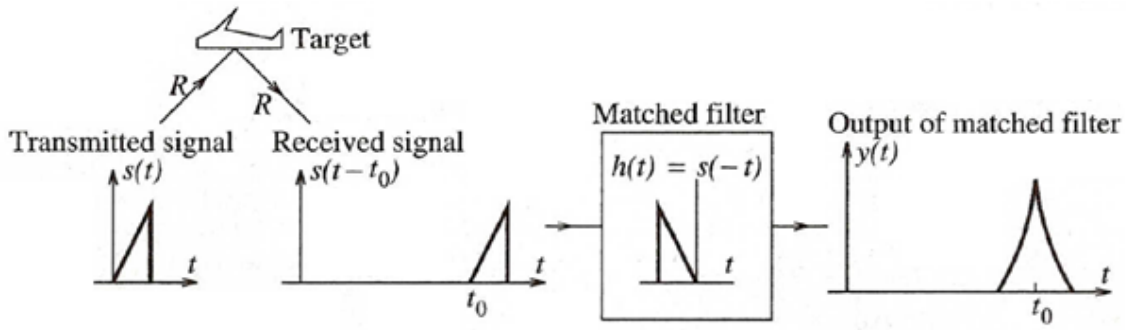

Figure 11: Matched filtering example[1]

2.2. Range Cell Migration Correction

The instantaneous slant range to the target changes as the radar platform passes a target, and thus the target is perceived to be at a different distance each time. The final image would be severely blurred in the range direction when signal compression was performed with the target spanning across several range bins. Therefore Range Cell Migration Correction (RCMC) is an important operation in SAR processing. The Range Cell Migration (RCM) with respect to azimuth frequency, f_η, in the range Doppler, range time and azimuth frequency domains, is shown in Equation 14. The approximation shown in Equation 14 is assumed in the simulation and is a close approximation, provided the squint angles are low. Equation 15 shows how to obtain azimuth frequency by using the azimuth FM rate, K_a.

$$R_{rd(f_\eta)} = \frac{R_{o_m}}{\sqrt{1-\frac{c^2 {f_\eta}^2}{4{V_r}^2 {f_O}^2}}} \approx \frac{\lambda^2 R_{o_m} {f_\eta}^2}{8{V_r}^2} \tag{14}$$

$$f_\eta \approx -K_a \eta \approx \frac{2{V_r}^2 \eta}{\lambda R_{o_m}} \tag{15}$$

Given that the migration must be calculated in discrete cells to be corrected for the RCMC process, the RCM is rounded to nearest integer for the simulation. In order to perform RCMC, the cells are shifted to counter RCM in the azimuth frequency domain prior to azimuth matched filtering.

3. PRIOR SAR IMAGING SIMULATION

As mentioned previously in this paper, the full 2-D SAR simulation and range-Doppler algorithm was developed in 2007 by Brian Zaharris. Zaharris' MATLAB simulation used the RDA to image point targets with limited mobility. Mason contributed to the development of the 2-D SAR simulation by implementing objects composed of arrays of point targets. However, Mason's code was not as efficient as Zaharris' code. Zaharris' MATLAB code also provided the components needed to design more complex two- and three-dimensional simulations. In June 2009, Matthew Schlutz used Zaharris' MATLAB code as a starting point for 2-D SAR simulations. Schlutz made modifications to Zaharris's code for more complex 2-D target simulations and also optimized the MATLAB algorithm for three-dimensional targets.

3.1. RDA example using Shultz's modified 2-D simulation

As an example of how the RDA defined in the previous section works, each of the intermediate stages are illustrated as follows, using Matthew Schlutz's simulation of a single point in the center of the target area.

The first step is to obtain the 2-D SAR signal space, shown in Figure 12, by using Equation 4 (see section 1.6 above). Figure 12 (below) shows the signal space with a jet color map where the lower magnitudes are represented with cooler colors and the higher magnitudes with warmer colors. The SAR echoes are shown spanning the entire azimuth and the center of the range. Also, AWGN can be seen throughout the image space. The Doppler effect causes the SAR echo energy to be spread over the azimuth, following a sinc-squared decay pattern near the points where the target lies in the center of the beam. The sinc-squared side bands are not visible. [5, 9]

Figure 13 shows a zoomed-in portion of the center of the signal space.

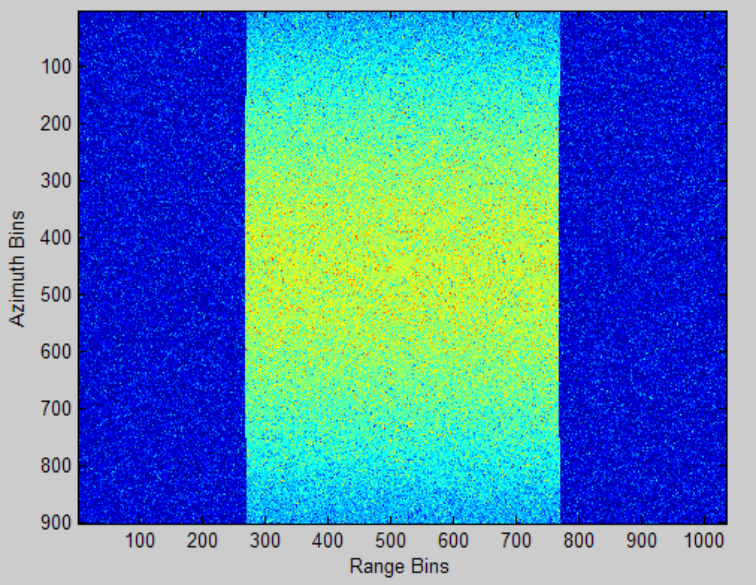

Figure 12: SAR signal space[5]

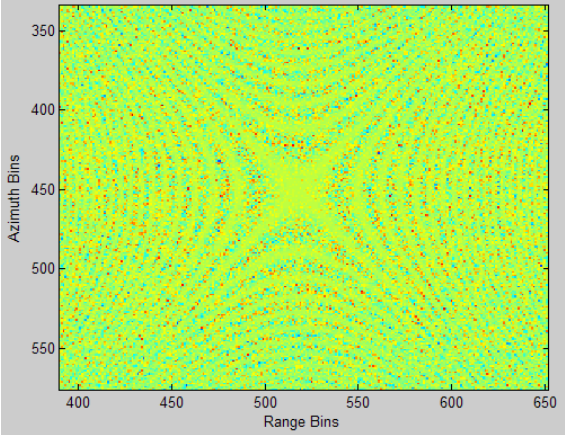

Figure 13: Zoomed-in center SAR signal space[5]

Once the SAR signal space is obtained, it goes through the RDA process shown in Figure 10 (see section 3 above). The first step in the RDA process is to perform range compression, which involves pulse compression by matched filtering. A range reference signal needs to be constructed in the range frequency domain, as shown in Figure 14 (below), in order to perform

matched filtering. The upper left section of Figure 14 shows the range time magnitude of the reflection, and the upper right corner shows the range time phase of the reflection. The lower left corner of Figure 14 shows the range frequency magnitude as a result of taking the Fourier transform of the reference signal, and the lower right corner shows the range frequency phase component.[5, 9]

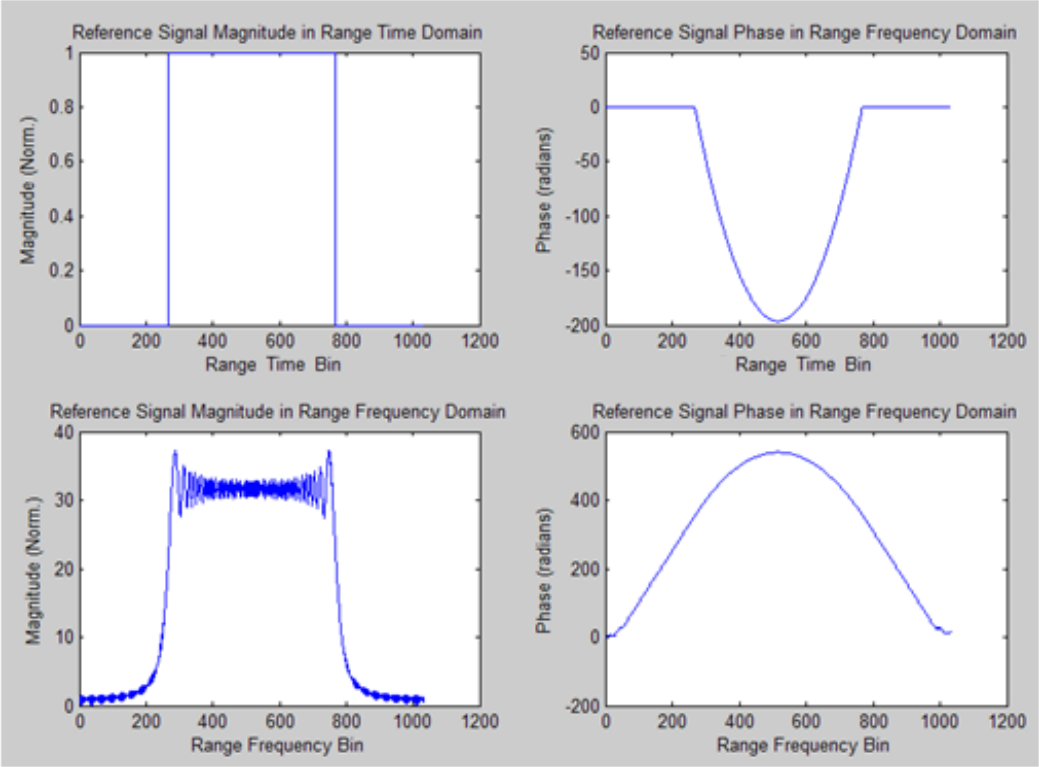

Figure 14: Range reference signal[5]

Once the reference signal is constructed, Equation 14 (see section 2.2 above) is used in the simulation to calculate the range cell migration. The results are shown in Figure 15 (below), and are rounded to the nearest integer because the range migration can only be corrected in discrete cells.

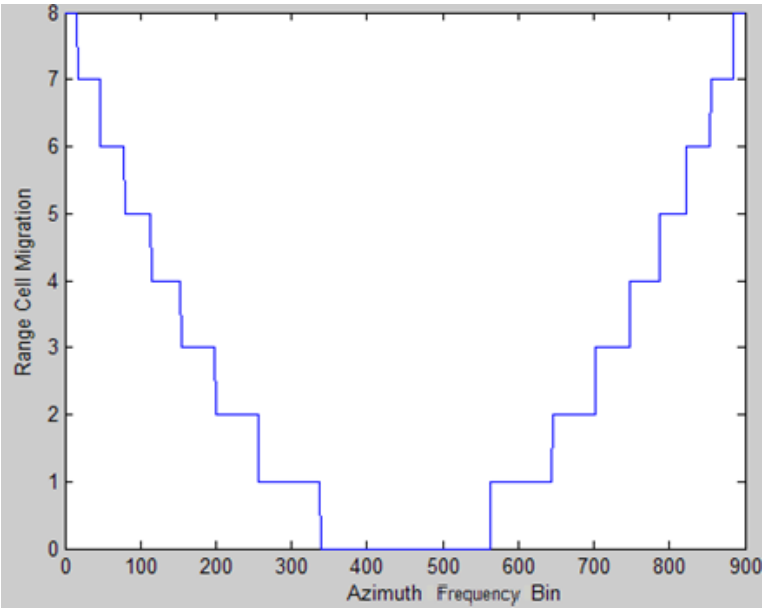

Figure 15: Range Cell Migration[5]

The range cell migration in this simulation is only valid when the target is within the regions of 400 and 500 bins of the azimuth. To deal with a target outside of this range, the RCMC shift should be recalculated to center around the range of azimuth bins that are desired.

The next step is to take the FFT of each range time signal and add AWGN to each signal. Figure 16 (below) shows in the upper left section the magnitude plot of an example range time signal obtained halfway through the flight. Although this signal contains AWGN, it looks very similar to the reference range time signal shown in Figure 14 (above). The upper right section of Figure 16 shows the FFT of the center azimuth range time signal.[5, 9]

After taking the FFT of the range time signal, the next step is match filtering (or convolution) of the FFT of the time reversed range reference signal (shown in the lower left section of Figure 14) with the FFT of the range time signal (shown in the upper right section of Figure 16). The result of the matched filtering is shown in the lower left section of Figure 16.[5,9]

The IFFT of the range matched filter output is then taken, resulting in a sharp and compressed pulse at the detected location of the reference signal, as shown in the lower right section of Figure 16. Although noisy backscatter can be seen in the image, this is negligible compared to the magnitude of the return from the target.[5, 9]

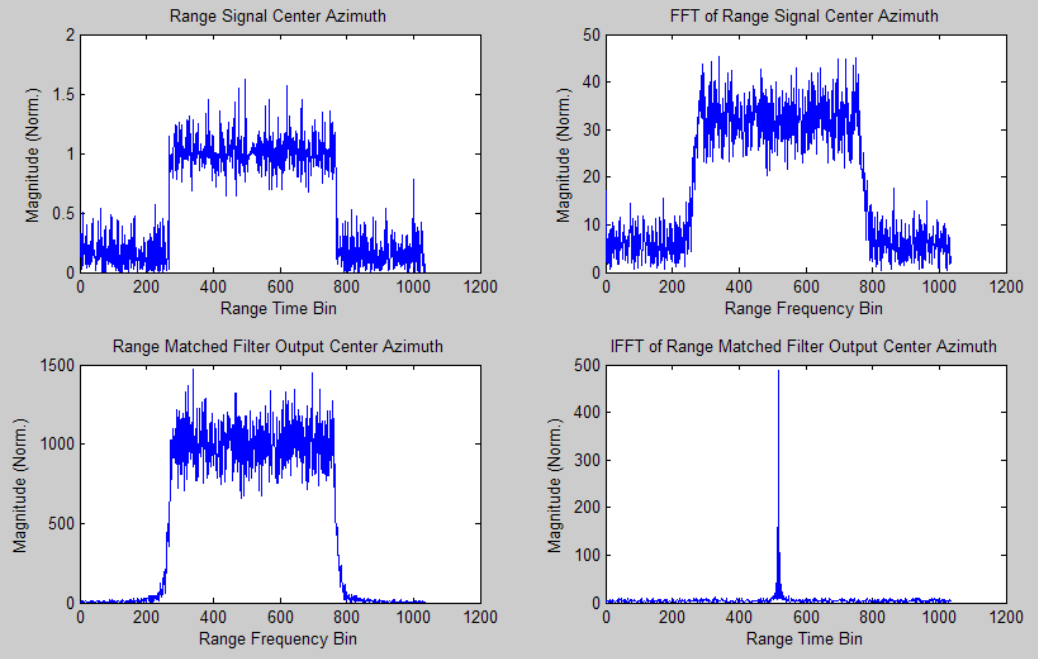

Figure 16: Compression example of range time signal at center azimuth[5]

The range compressed signal, once range compression is performed for each pulse, is shown in Figure 17.

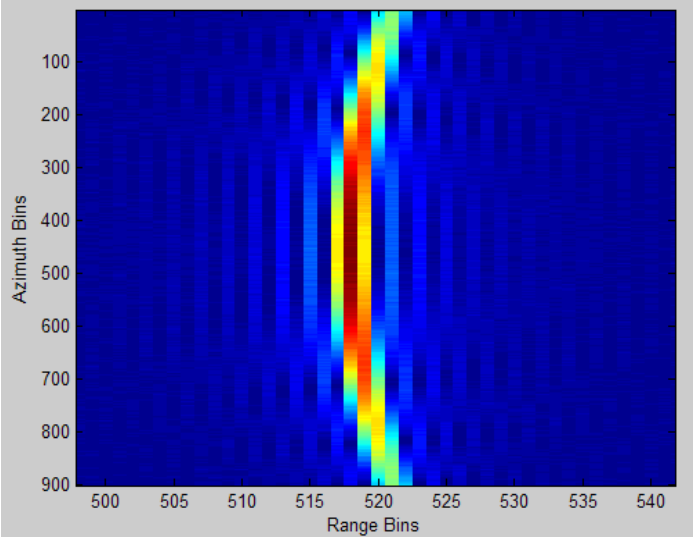

Figure 17: Image of range compressed signal[5]

The range cell migration that was predicted before the RDA filtering (shown in Figure 15) can be seen in Figure 17 (the image is zoomed-in in range to emphasize the range cell migration).

The next step in the range Doppler algorithm is to take the Fourier transform of the signal with respect to azimuth time. The main objective is to transform the azimuth time into the frequency domain while not affecting the range time. In this case, the Fourier transform is performed on each range bin. Figure 18 (below) shows the result of taking Fourier transform on the center range bin, whereby the spectrum becomes a function of Doppler frequency (also called azimuth frequency). As depicted in Figure 18, at zero Doppler frequency the magnitude of the spectrum is the largest, which corresponds to the half-way point of the flight path. The result obtained from combining the Fourier transform of each range bin is shown in Figure 19 (below). Range cell migration occurs in Figure 19 and thus it must be corrected in this domain.[5, 9]

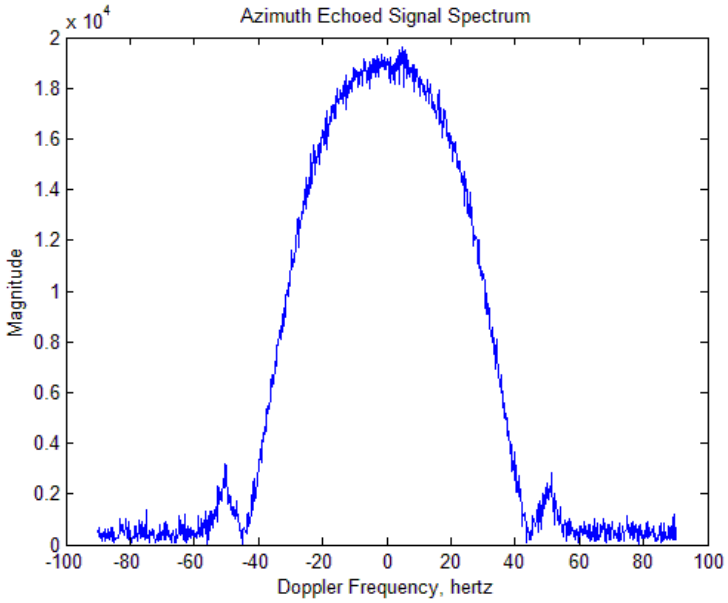

Figure 18: Result of the Fourier transform on the center range bin[9]

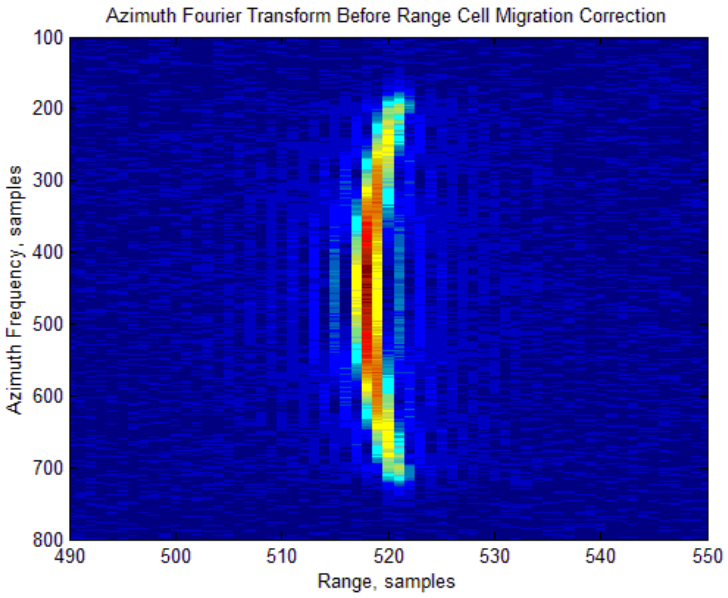

Figure 19: Azimuth Fourier transform for all range bins[5]

RCMC in the range-Doppler domain is performed before azimuth matched filtering. Figure 20 shows the image after RCMC is performed. The predicted shift shown in Figure 15 (above) is used to shift the energy along the azimuth to counter cell migration in the range direction.[5, 9]

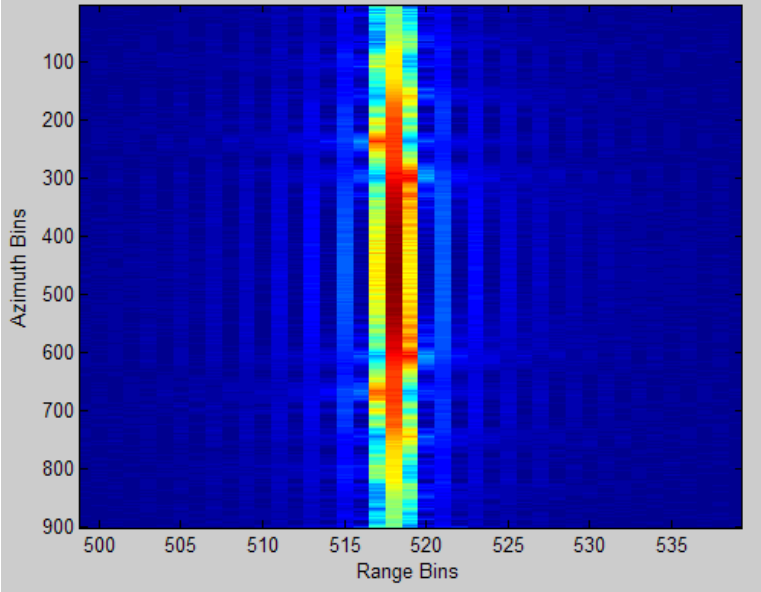

Figure 20: Image after RCMC[5]

After RCMC is completed, the azimuth matched filtering process can be performed. The steps for azimuth matched filtering are similar to those used for the range matched filtering process. A reference signal in the azimuth direction is first defined. The main objective is to compress the azimuth portion of the signal while keeping the range portion unchanged. Since azimuth matched filtering is done in the frequency domain, the reference signal needs to be Fourier transformed. The reference signal and the azimuth data, both in the frequency domain, are then convolved. The final step is to take the inverse Fourier transform of the result obtained from the azimuth matched filtering. After azimuth compression is performed on all range bins in the footprint, the final target image can be constructed as shown in the Figure 21 (below).[5, 9]

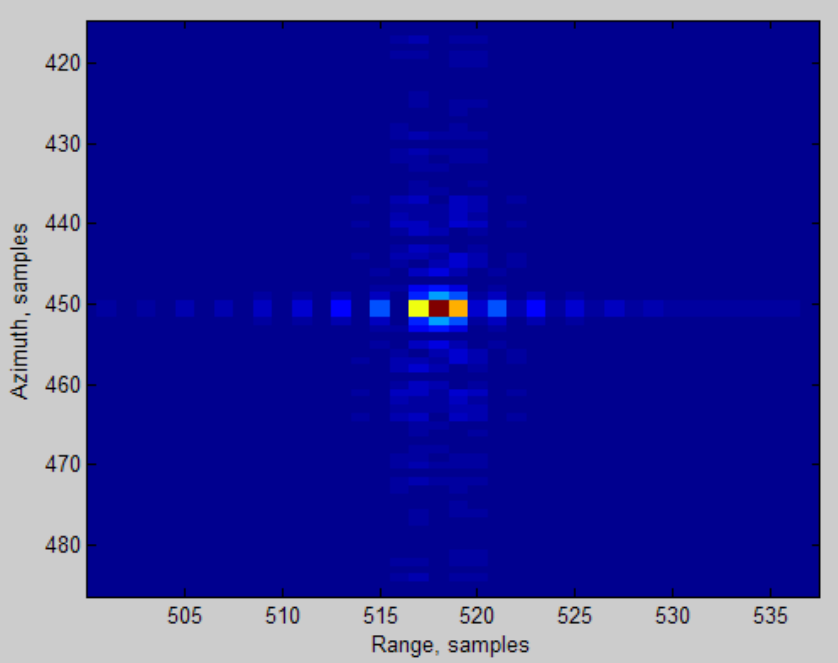

Figure 21: Final single point target image[5]

3.2. Shultz's improvements to Zaharris' simulation

Matthew Schlutz identified limitations in Zaharris' 2-D SAR imaging simulation, which Shultz considered would need to be addressed in order for its application to ATR to be more realistic. Schlutz modified the 2-D point target simulation by removing the target tracking feature in order to reduce memory usage. Schlutz optimized the echo generation sequence to reduce the processing time and added the capability of taking in a 16 by 16 target image by loading a bitmap of that size. Schlutz also developed a simple 3-D SAR simulation by importing multiple 2-D azimuth/range profiles at different altitudes. [5, 9]

Shlutz's 3-D SAR simulation required a new coordinate system that included altitude, as shown in Figure 22 (below). For this simulation, Schlutz assigned a location for the point targets and a constant altitude, $\boldsymbol{Z_o}$, for the platform. The 2-D range equation was replaced with a 3-D version, shown in Equation 16 (below). As shown in Figure 22, the variables that define the 3-D geometry are the look angle, $\boldsymbol{\theta_l}$, and the range of closest approach, $\boldsymbol{R_{o3D}}$. Also labeled in Figure 22 is the squint angle, $\boldsymbol{\theta_{sq}}$, which is defined as the angle between the slant range and zero-Doppler plane. [5, 9]

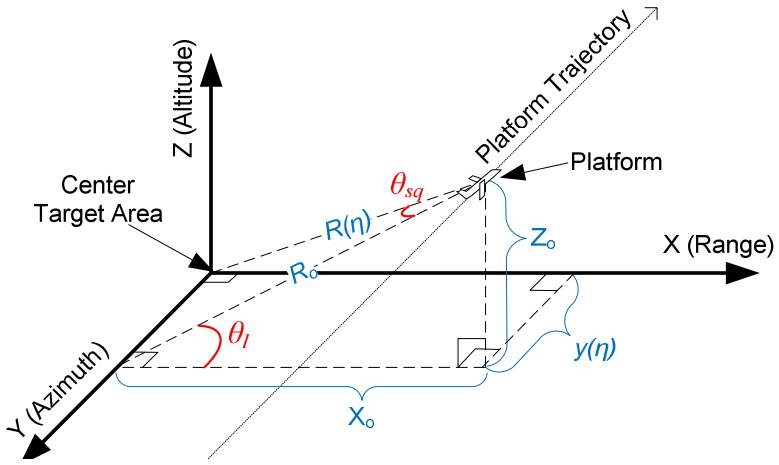

Figure 22: 3-D SAR geometry[5]

$$\boldsymbol{R(n)_{3D}} = \sqrt{\boldsymbol{R_{o_{3D}}}^2 + \boldsymbol{V_p}^2\, \boldsymbol{n^2}} = \sqrt{(\boldsymbol{X_o} + \boldsymbol{x_m})^2 + (\boldsymbol{Z_o} + \boldsymbol{z_m})^2 + \boldsymbol{V_p}^2 \left(\boldsymbol{n} + \frac{\boldsymbol{y_m}}{\boldsymbol{V_p}}\right)^2} \quad (16)$$

$$R_{o_{3D}} = \sqrt{X_o{}^2 + Z_o{}^2} \quad (17)$$

$$\tan(\theta_l) = Z_o / X_o \quad (18)$$

Shlutz also modified the RDA to reflect a 3-D SAR simulation by adding the constant altitude parameter, Z_o, in the range reference signal, azimuth reference signal, and range cell migration correction equations.[5]

3.3. Limitations of prior two- and three-dimensional SAR simulations

Upon starting this research project in Spring 2009, there were several areas that needed to be improved in order to have a SAR imaging simulation with which realistic SAR images could be generated for use in ATR testing and development. One of the major areas that needed attention was having larger images to improve the ability of the simulation to be used for ATR. The size of the input images used for the existing simulation was 16 by 16 pixels, thus limiting the distinguishable features that could exist in the target profiles. The SAR simulation took ten minutes to process a 16 by 16 pixel image. The simulation also contained an image import feature that was used to specify the location and reflectivity of point targets. The import images were 8-bit or 256 intensity level grayscale images and the intensity level was normalized to produce point target reflectivity. The images used were designed in Adobe Photoshop and saved in 8-bit grayscale GIF format. Most of the processing time was spent only in the SAR echo generation sequence, which presents a major impediment for application towards ATR. A

method for efficiently generating reflections from a larger set of point targets was needed. Also, the radar echo generation sequence needed to be optimized to improve the processing time.

Another major limitation of the two- and three-dimensional SAR MATLAB simulation was that it assumed that the point targets reflected at all times regardless of the line of sight visibility, and therefore the simulation was only realistic where line of sight was clear. The resolution was also very low and therefore the simulation was not useful for large scale image applications.

4. OPTIMIZATIONS MADE IN THIS RESEARCH PROJECT

As part of this thesis project, four major optimizations were made to improve the 3-D SAR simulation; obtaining range and reflection information using Blender, streamlining the antenna pattern, improving the echo generation, and using different programming languages.

In the new optimized simulation, an image of the target plane is rendered in Blender and then subsections of that image are processed by the echo generator program. The echo generator program sweeps the optimized antenna pattern across the entire Blender image to simulate the radar platform moving within the desired range. The final echo is then created by using the antenna pattern, the image reflectance and range obtained from the image rendered in Blender, and other user defined parameters. The echo data is saved as an OpenEXR file to then be loaded into Matlab. When the OpenEXR file is read into MATLAB it takes all of the settings entered in the previous GUIs and uses them to obtain the final SAR image. Each of the optimizations are explained in further detail below.

4.1. Image rendered in Blender

The range and reflectance of each point target were hardcoded in previous theses and therefore major research was undertaken to find a more realistic alternative. A 3-D graphics application called Blender was used to determine if a point target is visible or not, and to obtain the range to the point target. Blender can be used for modeling, texturing, animating, and rendering, among other uses. Blender takes image threads, renders the threads individually and finally merges all of those rendered threads back together. The target scene was created in Blender and the Blender camera and lamp was used to determine the plane within which the platform was moving (see Figure 23 below). Blender has a workspace limit of 10,000 Blender units in any direction.

Therefore, in order to simulate a radar platform with a range of closest approach of 20,000 meters to the center of the target area, a range scale factor was defined between Blender units and distance. Blender also allows the user to give material reflectance values to the models created.

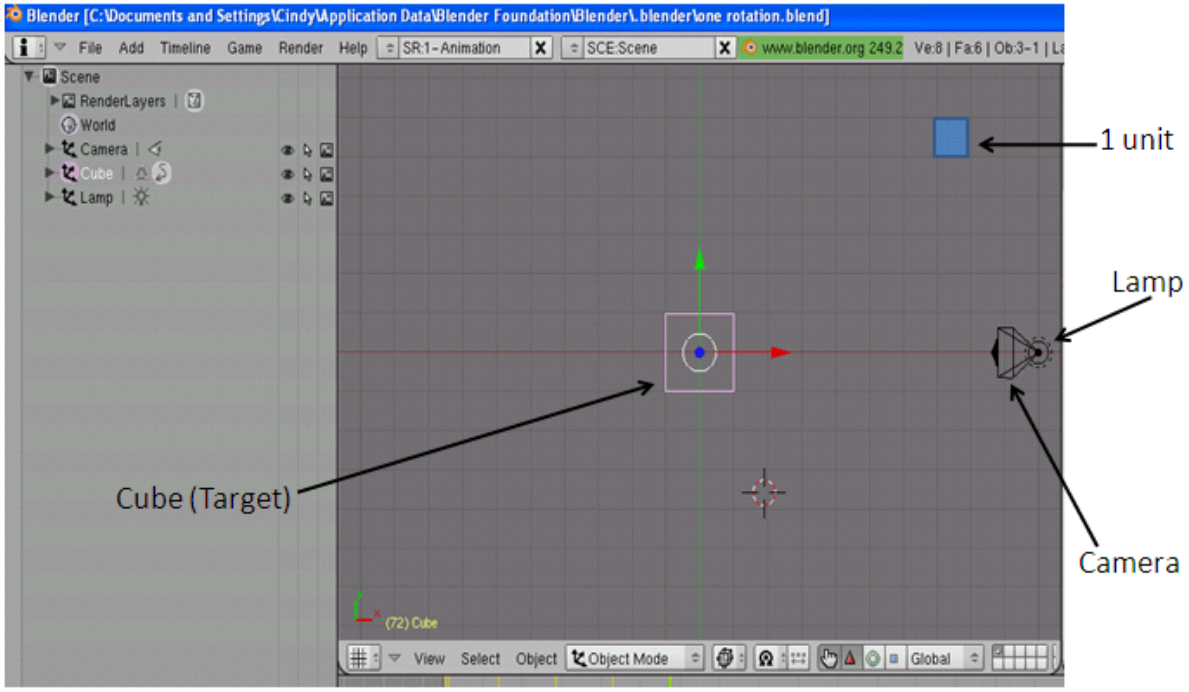

Figure 23: The Blender target scene

Blender supports two types of projection, namely orthographic and perspective projection (see Figure 24 below). Perspective projection is similar to the perception of a human eye, in that distant objects appear to be smaller than closer objects. Orthographic projection is where the axis or plane of the object is parallel with the projection plane.[7]

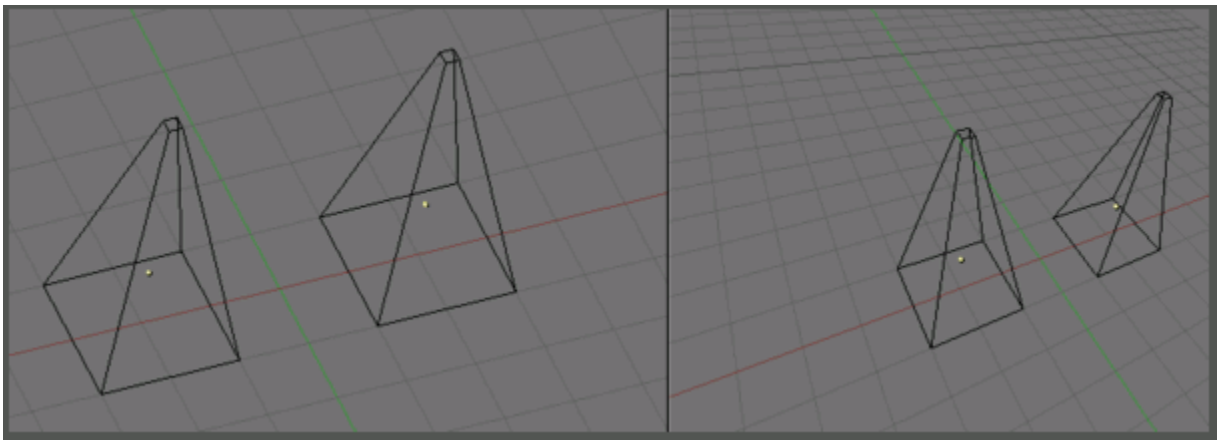

Figure 24: Perspective (left) and orthographic (right) projection[7]

Orthographic projection can only be used when running a simulation in stripmap SAR mode because the perspective of the camera does not move (whereas the perspective does change in spotlight and scan modes).

One of the useful features of Blender is that it allows the user to save the scene created and bring in different target models. Blender saves the rendered scene as an OpenEXR file. The OpenEXR library is developed in C++ and supports 16-bit floating points, 32-bit floating points and 32-bit integer pixels. A simple program was created in C++ that opens the OpenEXR file and extracts the Blender range and reflection data.

Blender is used to render a 7800 by 5000 pixel image and the Blender camera is set up in orthographic mode. By running the simulation in orthographic mode the perspective of the platform is not used for every single echo. Sub-frames from that 7800 by 5000 image are processed by the antenna pattern to simulate SAR. Taking sub-frames from a larger image significantly reduces the computer processing time. Since the antenna pattern only covers a maximum squint angle of 2.86 degrees, the amount of reflectance missed because of the simulation not running in perspective mode was calculated to be negligible, and therefore it was

not considered necessary to account for this in the simulation. The angle of incidence is so large that reflection is not likely to return to the platform.

At the start of this project, 900 individual images were rendered in Blender in order to obtain the perspective of the radar platform as it flies by the target. The method of rendering 900 individual images produced a variation in the target location and reflection, and therefore suboptimal simulation results caused by the inability of the matched filter to operate properly in the azimuth direction (see Figure 26). The received echo created by the echo generator program also shows results that are inconsistent with what would be expected from a single target in space before it is run through the RDA. The echo generated from processing 900 image shows distortion across the center as opposed to the echo generated by processing one large image where the star pattern in the center is clearly visible (see Figure 25). It also took approximately 1.5 hours to render the 900 images, compared to the 30 seconds it took to render 1 large image. Therefore the first approach was disregarded (code for this approach is available in Appendix E).

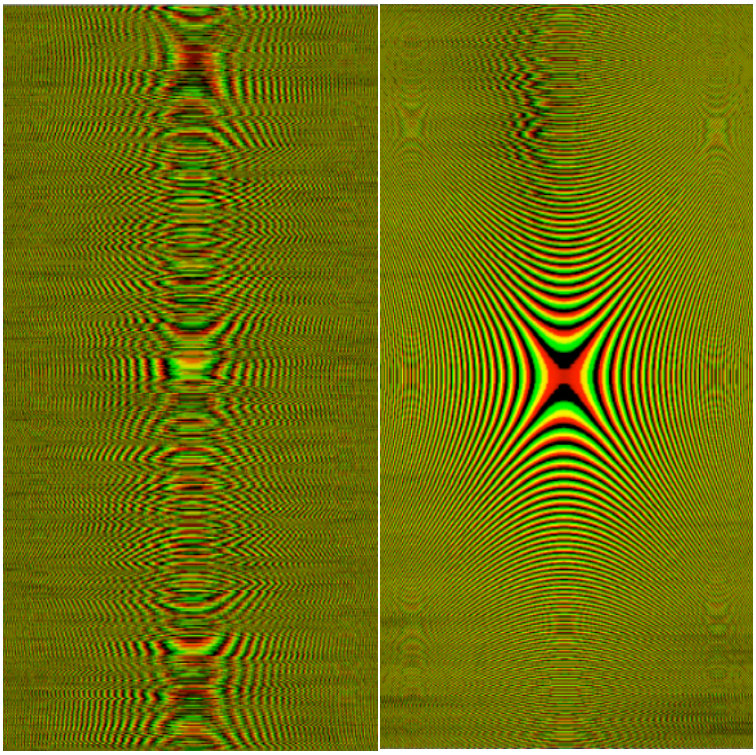

Figure 25: Left – echo generated from processing 900 images of a single cube in perspective mode; Right – echo generated from processing 1 image of a single cube in orthographic mode

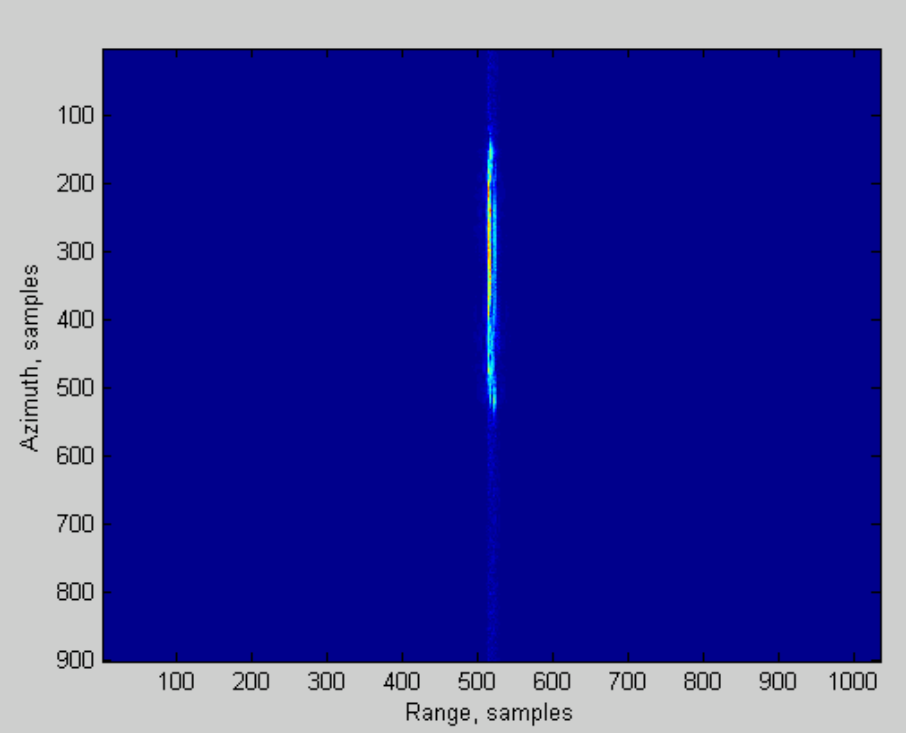

Figure 26: Inaccuracy of matched filter in the azimuth direction

(i) Blender Range

Blender creates a channel in the OpenEXR file named ‘Z’, which contains a floating point value that represents the range to each pixel of that image. The range data that Blender outputs is the distance from the platform plane to the target plane and therefore the data needs to be adjusted to obtain the range from the platform's precise location (within the platform plane) to the target's precise location (within the target plane). To calculate the actual range from the platform to the target, the range obtained from Blender is multiplied by the range modifier. The range modifier is calculated by multiplying the inverse cosine of the radial angle by the Blender scale factor (real world unit size assigned to a Blender unit) (see Figure 27 below). The operations are minimized by multiplying the Blender scale factor in the range modifier calculation, as opposed to multiplying it in the actual range calculation (which is calculated once per pixel for every echo). For the purpose of this thesis the radial angle is defined as the angle between the boresight and the vector between the platform and the current target location. Calculations for the radial angle are explained in further detail in section 4.2. The range modifier is then multiplied by the Blender range in order to obtain the actual range from the platform to the target. The range calculated is a very close approximation of the actual range. By calculating an approximate range, processing efficiency was improved by not recalculating the actual range on a per pixel basis.

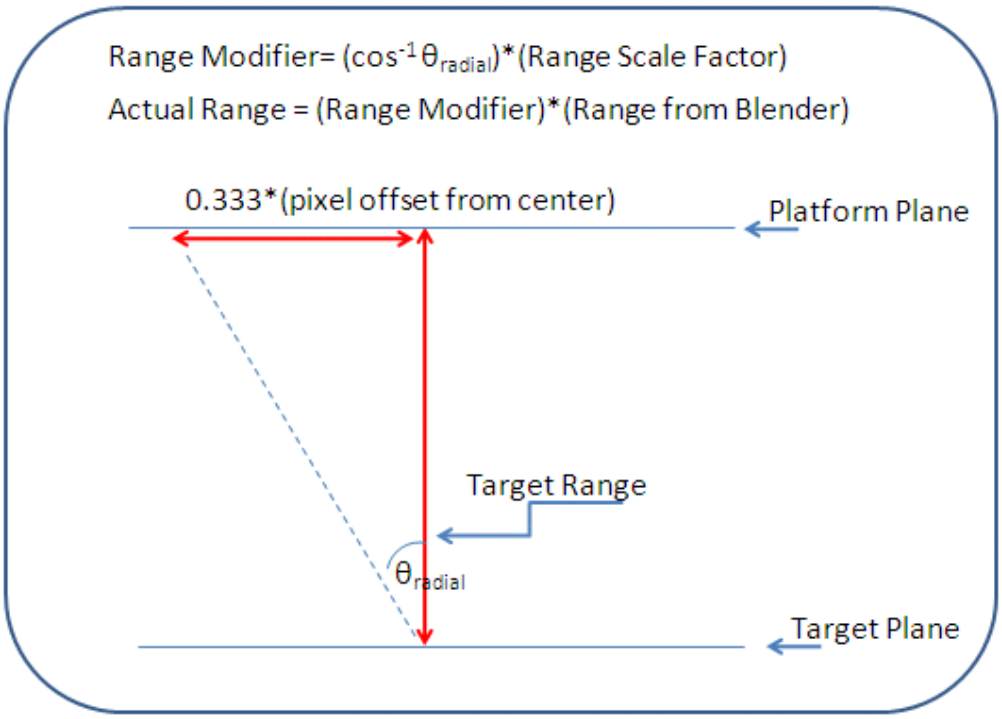

Figure 27: Trigonometry used to find the actual range

(ii) Blender Reflectance

Blender also gives reflection information about each pixel by creating channels named 'R' (Red), 'G' (Green), and 'B' (Blue) in the OpenEXR file. The higher the value that each channel has, the higher the reflection.

In this thesis project, the reflectivity of the target, F_n – which is used in Equation 4 (see section 1.6 above) – is represented by the pixel reflectivity given by Blender.

4.2. SAR antenna pattern

The antenna pattern used for past theses, and for this thesis, is based on a sinc equation as a function of where the antenna is pointing. The antenna pattern algorithm used previously

recalculated the antenna's input (w_a) for each target at every single echo generated, therefore performing a significant number of lengthy recalculations. In order to reduce processing time, a method was found where the calculation of w_a could be made more efficient.

In this thesis, the antenna's input was pre-calculated for thousands of points within the target plane. Then, for each echo generated for each target, once the target's location within the target plane was identified, the pre-calculated w_a value corresponding to that location was used. By taking advantage of the fact that the w_a value for each point within the target plane could be accurately pre-calculated, and by assuming that the camera axis is at the center of the image, the need to perform antenna input recalculations for every echo from every target has been avoided.

A C++ program was created for the new antenna pattern method. The program calculates the angular offset (or radial angle) for any given pixel in the image from the center of the target plane. Factors used in this calculation include: the size of the antenna pattern; the pixel separation (distance between two adjacent pixels); the antenna length, the signal frequency, the range scale (scale factor defined for a Blender unit); and the target range (range of closest approach from the platform to the center area of the target). The number of pulses per second was obtained by dividing the total number of pulses by the flight duration desired for the simulation. The platform movement per pulse was calculated by dividing the platform velocity by the number of pulses per second. The pixel separation was then calculated by dividing the platform movement per pulse by antenna pattern movement.

The antenna pattern program calculates the radial angle by dividing the target offset by the target range and then taking the arctan of that term (see Figure 28). The pixel offset is defined by the number of pixels in the x and y direction that the target is offset from the center axis of its plane.

The distance of the pixel offset in the x and y direction are calculated by multiplying the pixel separation by the number of offset pixels in that direction. The distance that the target is offset from its center axis is finally calculated by taking the x and y direction pixel offset and using trigonometry (see Figure 29 below). Once the radial angle for any given pixel in the image is calculated, it is then used in Equation 8 to obtain w_a.

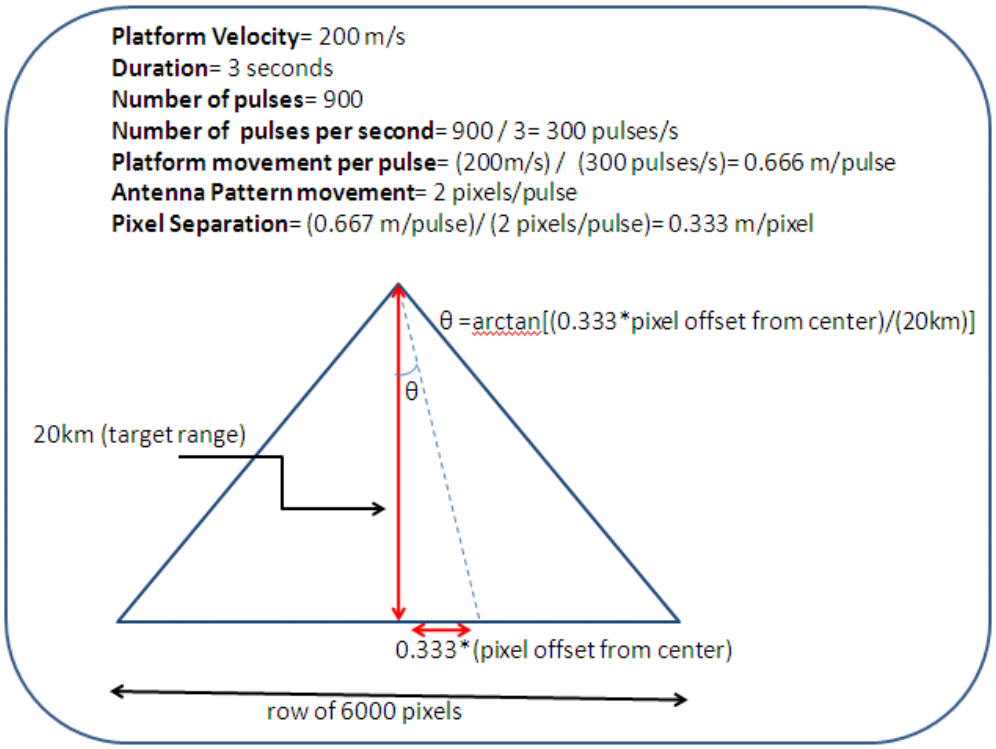

Figure 28: Trigonometry used to find the angular offset (radial angle)

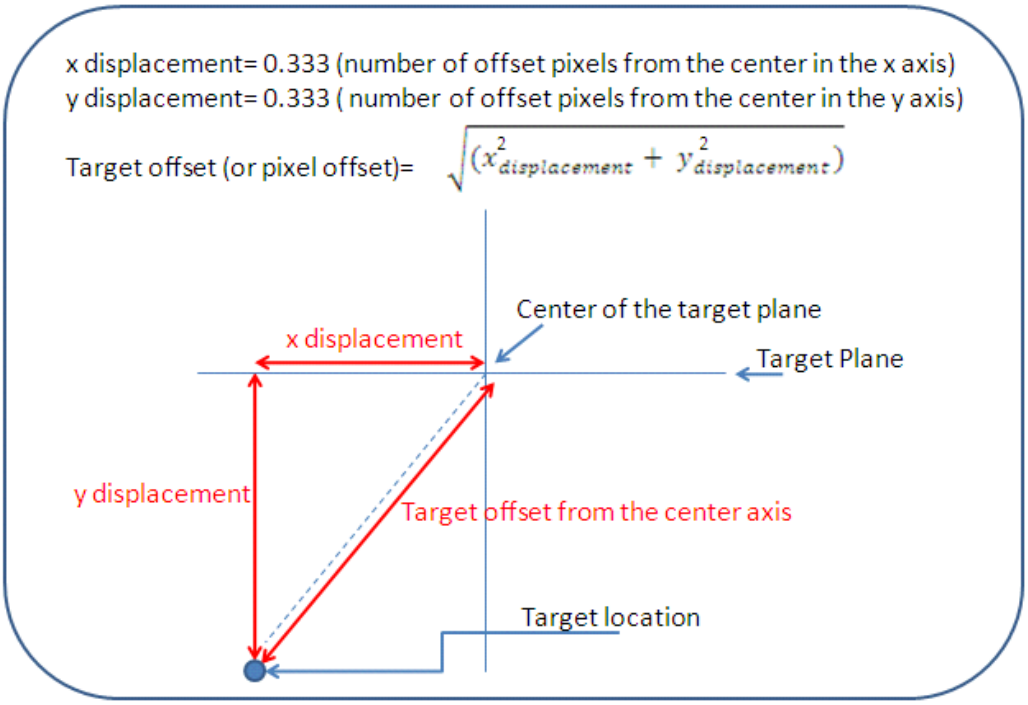

Figure 29: Trigonometry used to find the platform offset

By calculating the antenna pattern once, it can be used for multiple simulations and the processing time is significantly reduced. To facilitate reusing antenna patterns, a program has been created in C++ that saves the antenna pattern in an OpenEXR file. The image format in the OpenEXR file is used to allow the antenna pattern to be read in by the echo generation program.

One of the benefits of the antenna pattern generator having its own program is that changes can be made to the antenna pattern without needing to re-run other parts of the simulation (so the image obtained from Blender would not need to be recreated).

4.3. Echo generation

The echo generation Matlab code used in previous theses was rewritten for this thesis project. The processing efficiency of the SAR echo generation was optimized by porting the code to C++

and pre-calculating the Quadrature Demodulation Signature (QDS) of the transmitted signal (rather than calculating it for every single target).

The echoes are created as a product of several coded and user-defined parameters. The coded parameters are the antenna radiance pattern, an image (obtained from Blender) with reflectance information, the modified Blender range, the QDS (calculated as shown below). The user-defined (or set-up) parameters, which the user must enter in the GUI, are the target area width, the pulse repetition frequency, the baseband bandwidth (±Bo), the pulse duration, the number of pixels per pulse and the number of pulses for the entire simulation. Once these parameters are entered, the GUI will automatically calculate the number of echo bins, demodulation samples and the platform velocity. There are three steps involved in the calculation of the number of echo bins. The first step adds 1 to the range delta, then divides that term by the speed of light, and finally multiplies the term obtained by 2. The second step multiplies the signal bandwidth by 2. The third and final step adds the result from the first and second step, and then rounds up the term obtained. The demodulation samples are calculated by multiplying the pulse duration by 2 times the signal bandwidth, then rounding up the term obtained, and finally adding 1 to the rounded term. The platform velocity is calculated by multiplying the total number of pulses in the simulation by the number of pixels per pulse and then multiplying that term by the pixel separation.

The user-defined parameters are also used to calculate the QDS, which for the purpose of this thesis project, was defined as set out in Equation 19.

$$\text{QDS} = e^{j\pi K_r\left(t-\frac{2R_m(\eta)}{c}\right)^2} \tag{19}$$

One of the major optimizations made to the echo generation code was to calculate the QDS only once for each location and range. The QDS is pre-calculated for the period where $t - \frac{2R_m(\eta)}{c}$ equals 0 to t_p (chirp duration), and the values outside of this period are set to 0. This property allows the QDS to be calculated outside of the loop because it varies based on time only, and does not vary based on location or range. Calculating the QDS outside of the loop also reduces processing time. The loop used in previous theses calculated the QDS for every target during each echo generated, thus making 230,400 QDS calculations (900 echoes multiplied by 256 targets).

To properly sample the demodulated data, the sample rate must equal twice the signal bandwidth. The number of samples obtained from the QDS of the transmitted pulse is defined by the time it takes to transmit the signal and the signal bandwidth.

The QDS array is a one-dimensional array containing the demodulation signature of the transmitted radar pulse, as shown in Figure 30.

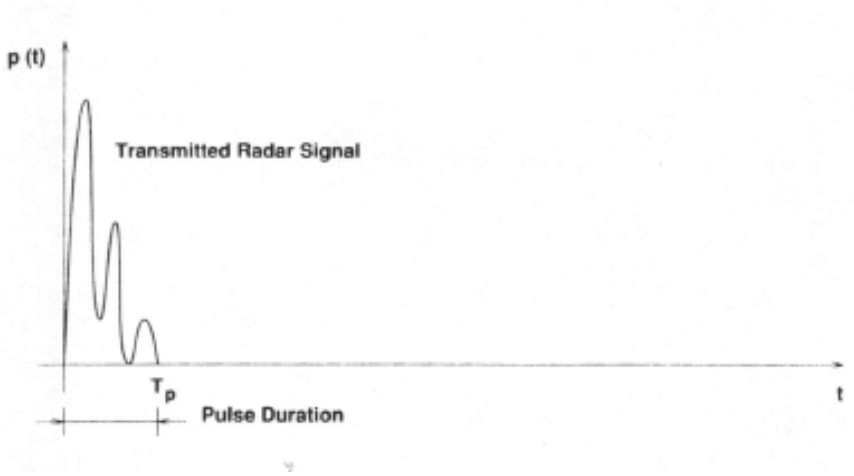

Figure 30: Transmitted radar pulsed signal[1]

The QDS array is then multiplied by a timeline as part of creating the echo signal. The timeline is defined by plotting the reflection magnitude at a point in time (based on range) for each target in the image. The reflection timeline is therefore given by placing each pixel in the image on the correct point along the timeline, as shown in Figure 31 below.

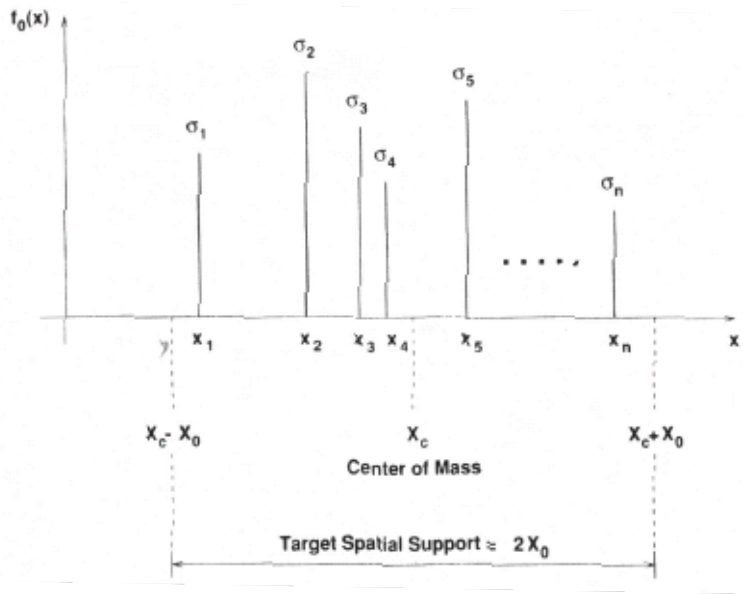

Figure 31: Target reflection timeline[1]

The reflection timeline is then multiplied by the QDS to obtain the echo signal (shown in Figure 32).

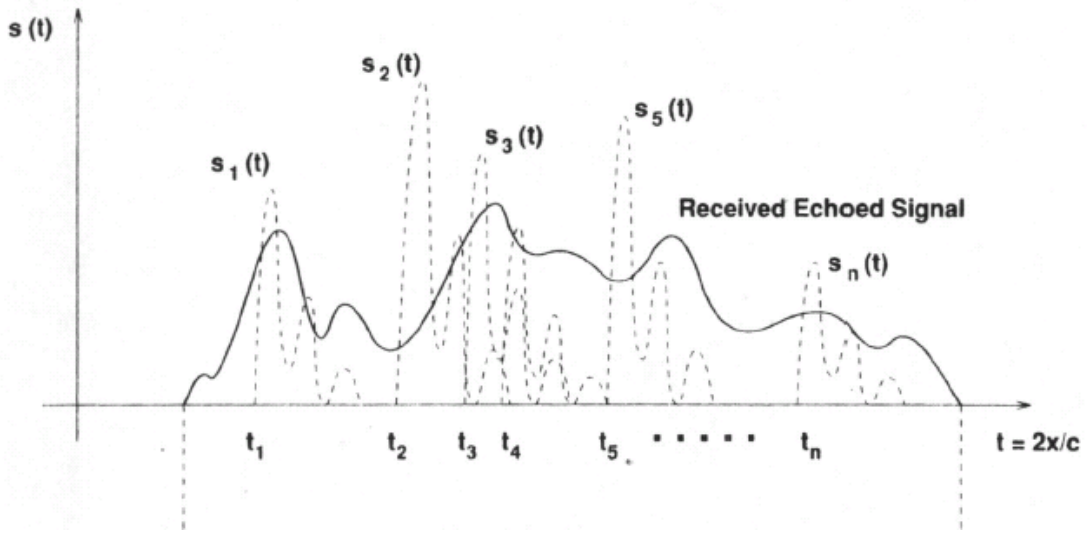

Figure 32: Received echo signal timeline[1]

4.4. Programming Languages

Most of the development of the simulation for this thesis project was done on a Macintosh (Mac) computer. The coding used to build the GUI and display was written in Objective-C. Objective-C is an object-oriented programming language which uses messages to establish communication between objects. One of the advantages of Objective-C is that it can directly interface with C++ objects and functions. The Objective-C portions of the program are not portable and therefore cannot be used on a computer that does not run a Macintosh operating system. However, only the coding that is related to the GUI and display was written in Objective-C, and this coding could easily be rewritten in other programming languages for use on different operating systems.

The OpenEXR libraries were available to use in C++, thus making it easier to interface to a C++ program. Other important algorithms such as the antenna pattern and the echo generator were written in C++.

Matlab was still used in this thesis to run the echoed signal through the RDA algorithm and obtain the final simulation images.

5. 3-D SIMULATION

There are four major steps involved in running the 3-D simulation developed in this thesis: obtaining a rendered image from blender; creating an antenna pattern using the antenna pattern GUI; creating an echo using the echo generation GUI; and running the echo through the RDA in Matlab to obtain the final image. Each of these steps are illustrated below by way of an example 3-D simulation.

Blender is set up to output an image of size 7800 by 5000 and the Blender camera is set up in orthographic mode with a 45 degree elevation angle. For the purpose of this thesis project, one unit in Blender was defined to be ten meters because rendering needs to be accomplished at 20 kilometers (2,000 units in Blender). The scene of the static target(s) and the radar platform was created in Blender. In this 3-D simulation example, a 2 by 2 meter cube was placed in the middle of the target plane (see Figure 33 below). A material reflectance of 1.0 was given to the cube. The Blender camera and lamp were placed at the same location because energy had to be radiated from the camera to simulate the radar platform. In order to obtain a resolution of 0.333 meters, the horizontal size of the blender image was set to 7800 pixels and the lens of the camera was set to a scale of 260 Blender units.

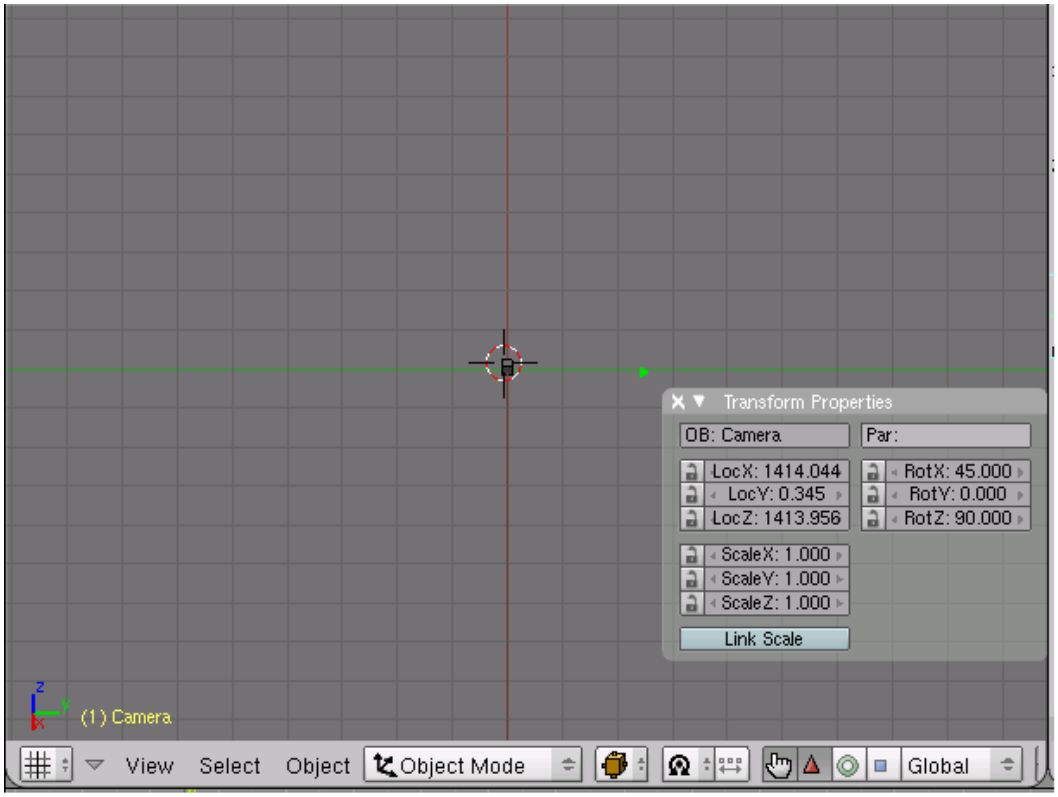

Figure 33: Blender scene – 2 by 2 meter box

Once the scene was created in Blender, it took less than 30 seconds to render a 7800 by 5000 pixel image. The image rendered in Blender was then saved as an OpenEXR image file.

Once the OpenEXR file was created, the antenna pattern was generated by opening its GUI and entering values for each of the parameters needed to create the antenna pattern (as shown in Figure 34). For this simulation, the antenna pattern was set to a height of 5000, a width of 6000, a pixel separation (resolution) of 0.333 meters, an antenna length of 2 meters, a frequency of 4.5 GHz, a range scale factor of 10 meters to 1 Blender unit, and a target range of 20 kilometers. Beneath the 'Save File' button, there is an exposure button that can be moved around to scale the data differently in the antenna pattern GUI. Once the parameters were entered, the GUI displayed the antenna pattern created, which was saved in order to use it in the echo generation GUI.

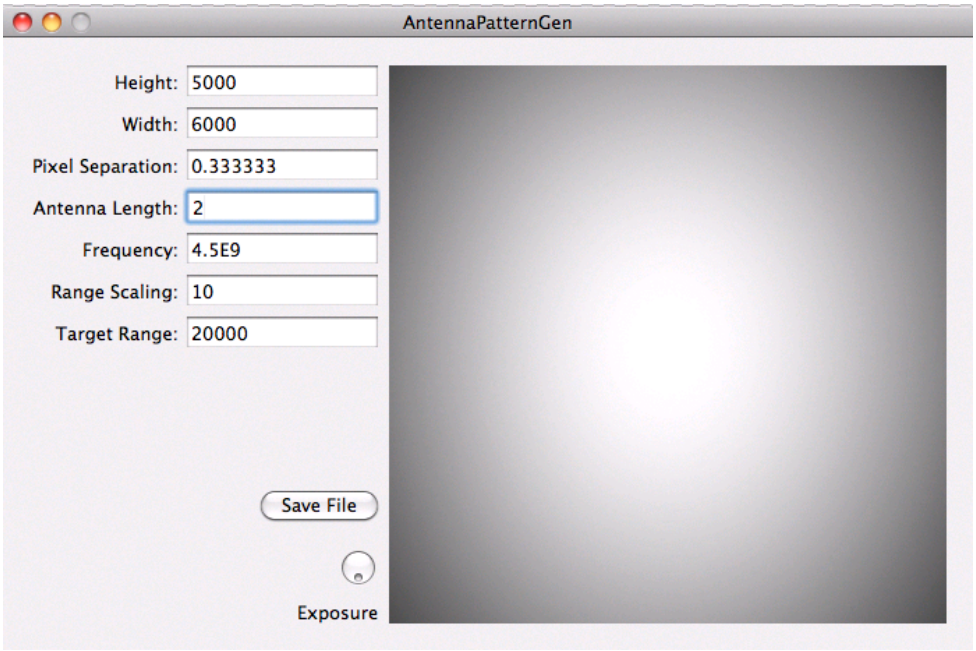

Figure 34: Antenna pattern GUI

Any changes to the parameters in the GUI result in changes to the antenna pattern. For example, comparing Figures 34 and 35 shows the changes to the antenna pattern resulting from amending the pixel separation input from 0.333 to 3 meters.

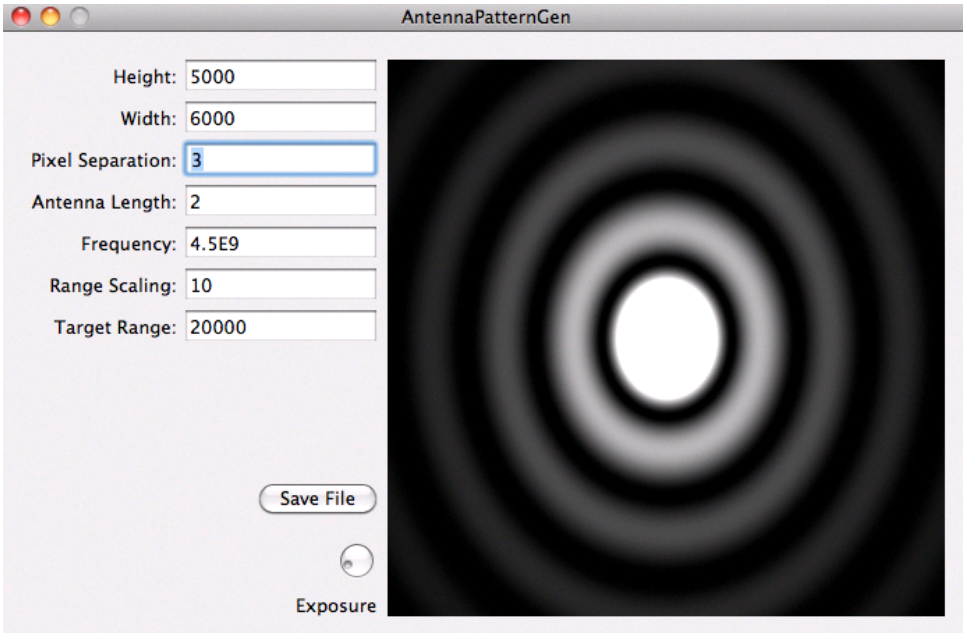

Figure 35: Antenna pattern GUI with a different pixel separation

Once the antenna pattern file was saved, the echo generation GUI shown in Figure 36 was opened and the simulation parameters for the echo generator were entered. For the simulations in this thesis, the echo generator parameters were set to; a target area (delta between the minimum and maximum range) of 400 meters, a pulse repetition frequency of 300 pulses per second, a signal bandwidth of 100 MHz, a pulse duration of 2.5 microseconds, a velocity of 2 pixels per pulse, and a total of 900 pulses for the entire simulation. Once these parameters were entered, the GUI automatically calculated and displayed the platform velocity, and the number of echo bins and demodulation samples. The saved antenna pattern file was then loaded into the echo generator GUI by pressing the 'load antenna pattern' button. Once the antenna pattern was loaded, the echo generator GUI displayed the height, width and carrier frequency of that antenna pattern.

The Blender rendered image file was then loaded in order to extract the reflection and range data (by pressing the ‘Load Range Data’ button in the GUI). The echo generation program then took the antenna radiation pattern, loaded the reflectance and range information from the Blender image file, and created an echo. The GUI then displayed the echo created (see Figure 36 below). Once the echo was created, it was saved as an OpenEXR file by pressing the ‘Save Echo Data’ button in the GUI.

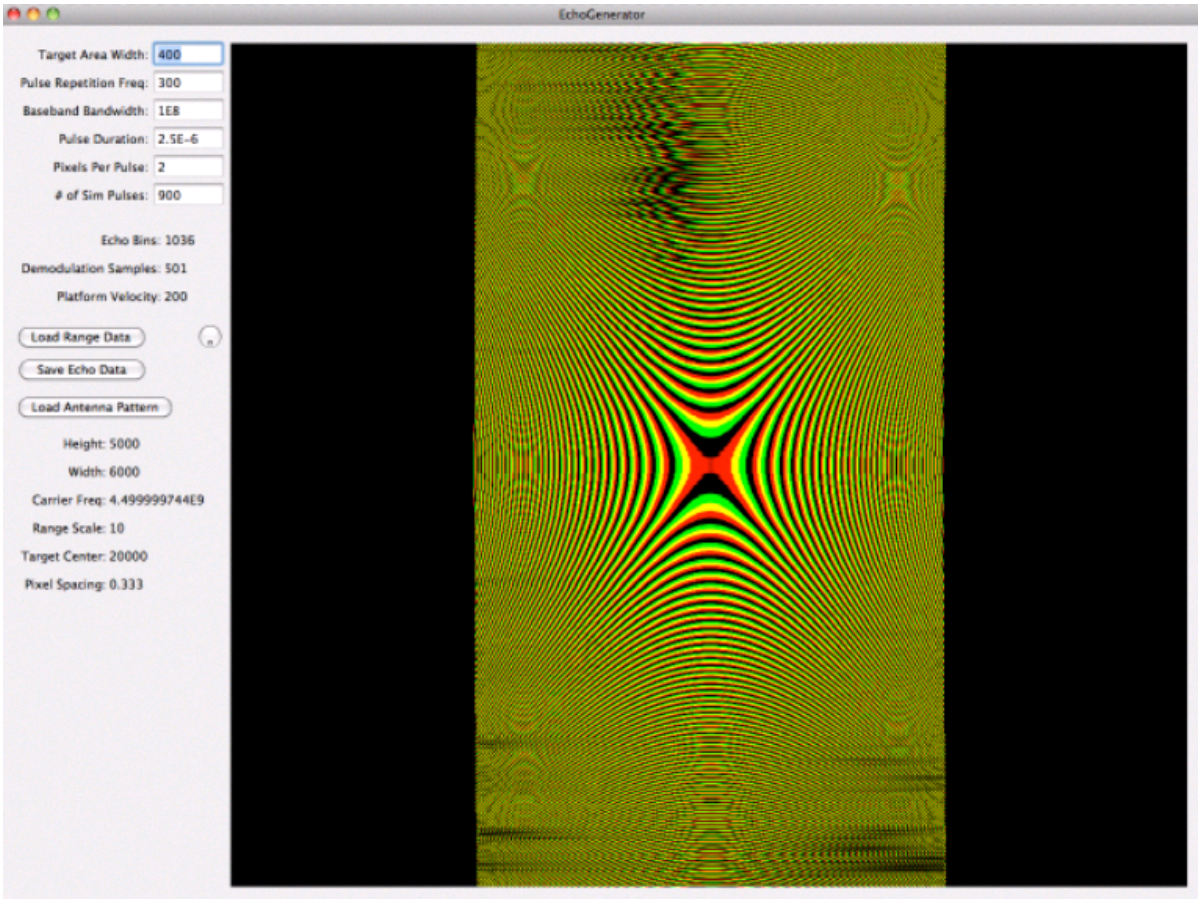

Figure 36: Echo generated – 2 by 2 meter box

The echo created then went through the RDA by calling the 'ExrEchoToRDA.m' file into Matlab. A dialog box then asked for the EXR file generated by the echo generator. Matlab ran the echo through the RDA and then displayed the range compressed SAR image (shown in Figure 37) and the final SAR image (shown in Figure 38).

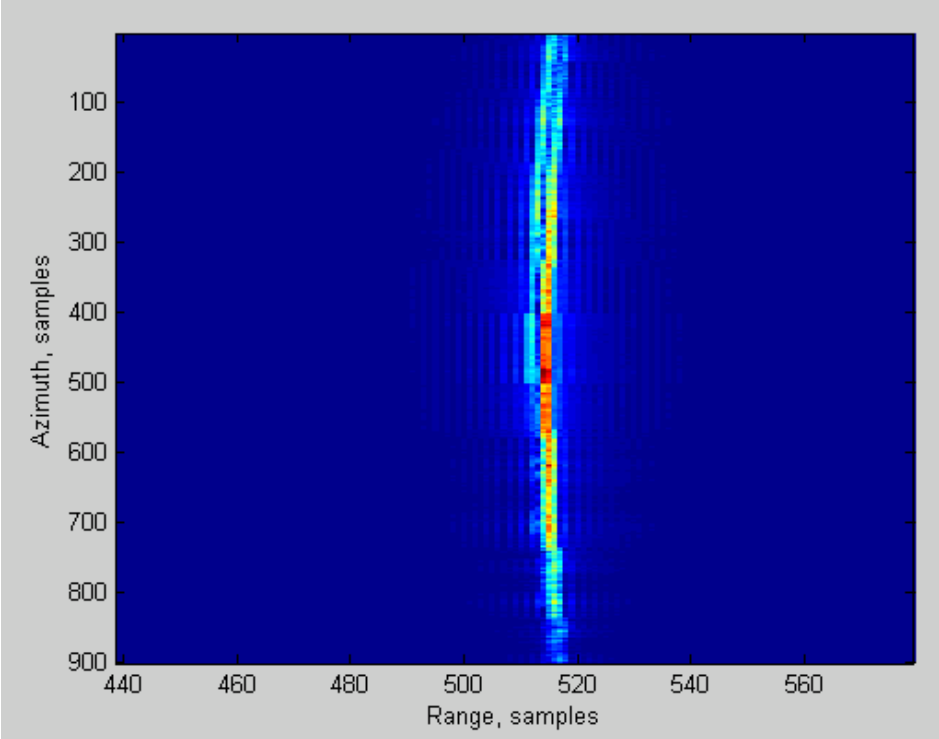

Figure 37: Range compressed SAR image – 2 by 2 meter box

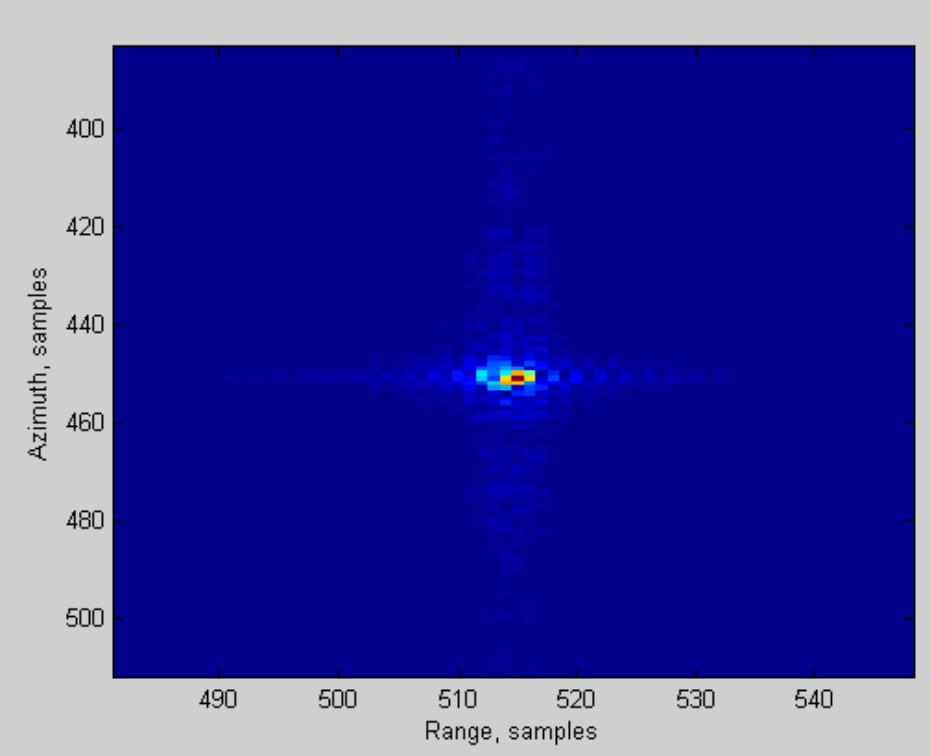

Figure 38: Final SAR image – 2 by 2 meter box

6. ADDITIONAL SIMULATION RESULTS

The results shown in Figure 38 were very encouraging because they demonstrated that the simulator is able to process a high resolution image. However, additional simulations were run of different target scenes to better test the full capabilities of the optimized simulator.

6.1 Azimuth and range resolution experiments

In order to test the accuracy and the limits of the optimized simulator, a scene with multiple cubes was created in Blender and the same simulation parameters as in section 5 were used. For the first experiment, 9 cubes were configured as shown in Figure 39. One cube was placed in the center of the scene, 4 cubes were placed 8 meters from the center cube, and 4 cubes were ambitiously placed 1 meter from the center cube. To ensure the cubes were distinguishable from each other in the final SAR image, a higher material reflectance value was given to the cubes placed 8 meters away from the center cube in the range direction than all the other cubes. The echo generated, the range compressed SAR image and the final SAR image are shown in the Figures 40, 41 and 42 respectively.

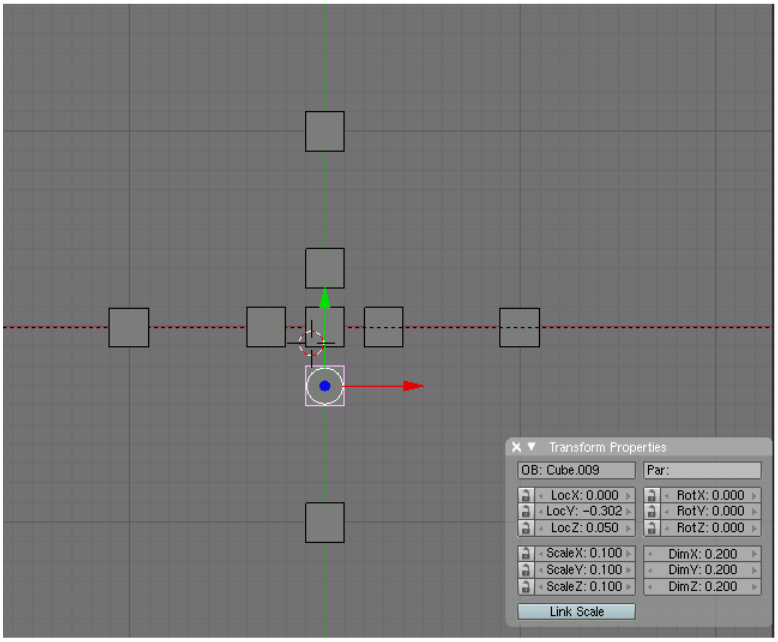

Figure 39: Blender scene – cubes 1 and 8 meters away from the centre cube in both directions

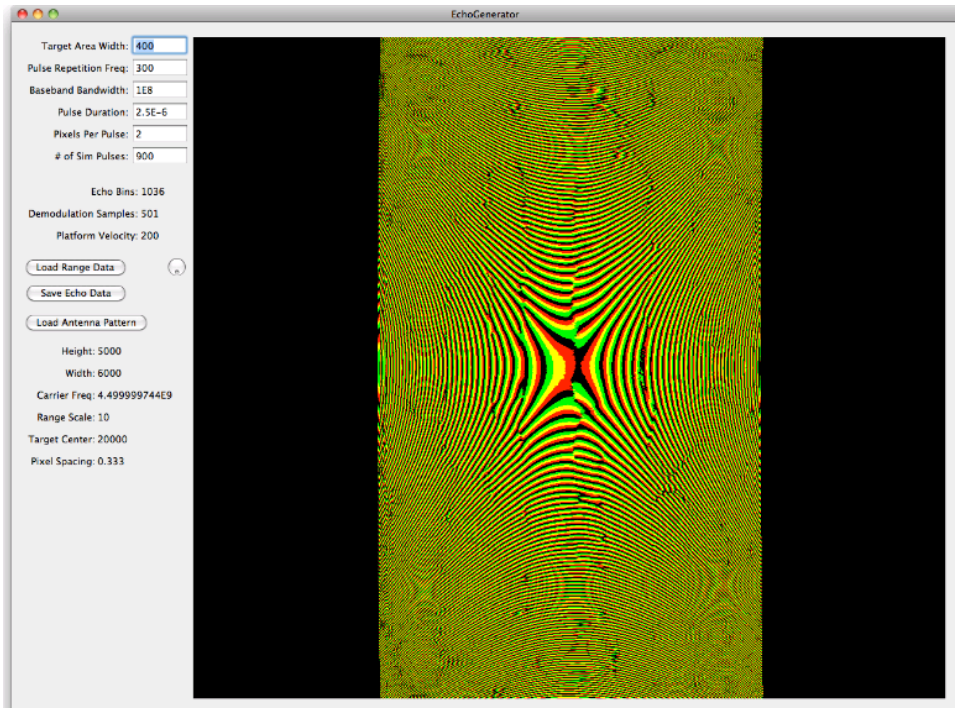

Figure 40: Echo generated – cubes 1and 8 meters away in both directions

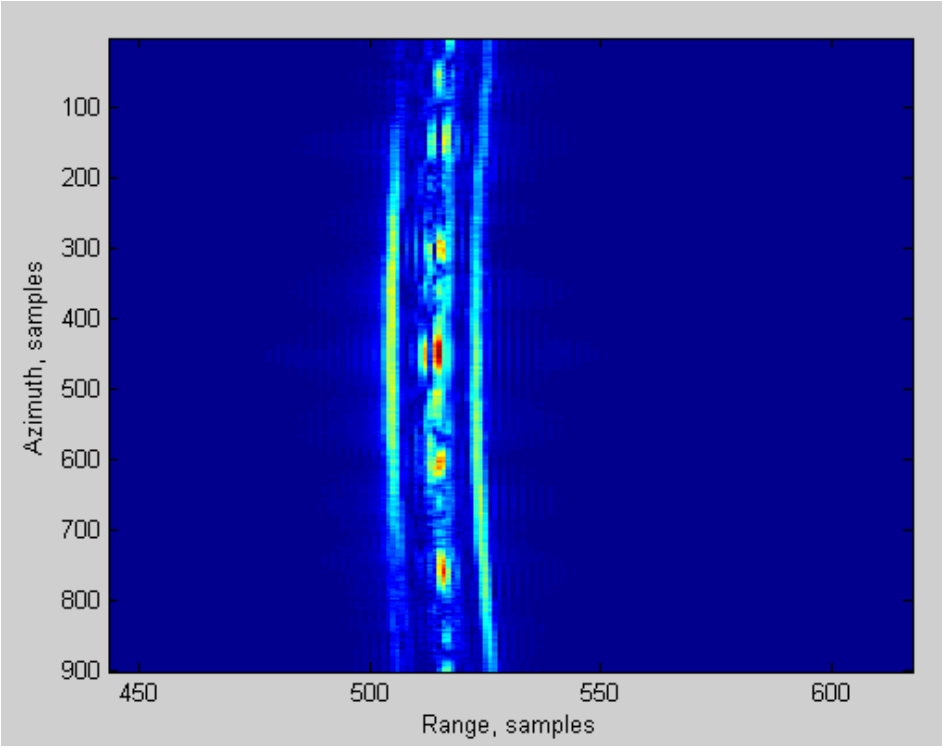

Figure 41: Range compressed SAR image – cubes 1 and 8 meters away in both directions

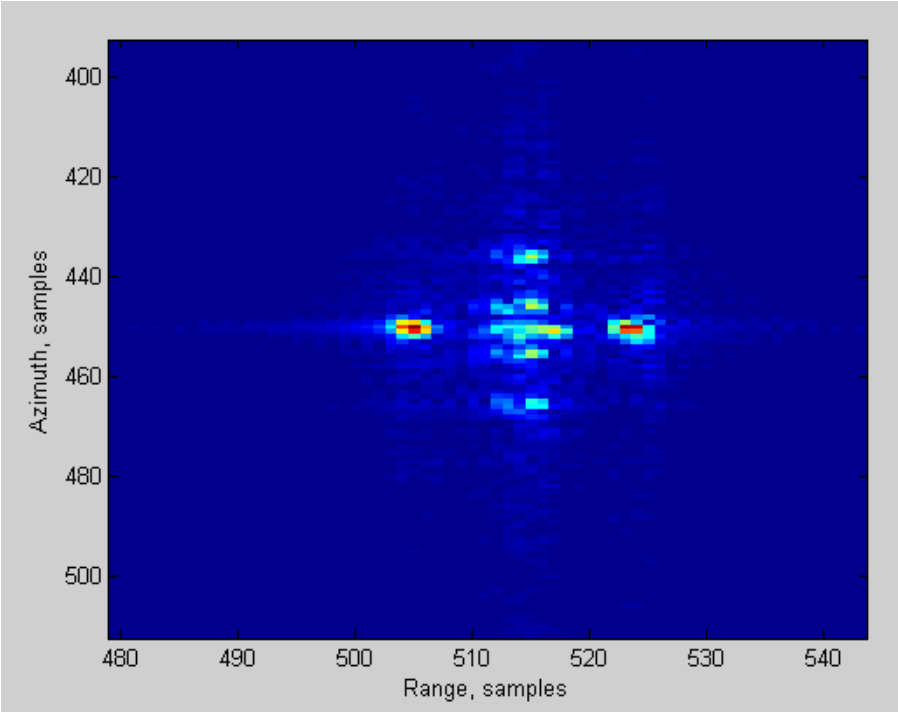

Figure 42: Final SAR image – cubes 1 and 8 meters away in both directions

The final SAR image above shows that the cubes that are 1 and 8 meters away from the center cube in the azimuth direction, and the cubes that are 8 meters away in the range direction, are clearly defined and visible. However, the simulation results show that when targets are placed within 1 meter of each other in the range direction, those changes are misinterpreted as changes in the azimuth direction, resulting in ambiguities in the range direction.

The cubes that were 1 meter away from the center cube in the range direction were moved to 2 meters away from the center cube in the range direction for the second experiment (see Figure 43). The echo generated, the SAR image after azimuth compression and the final SAR image are shown in the Figure 44, 45 and 46 (respectively).

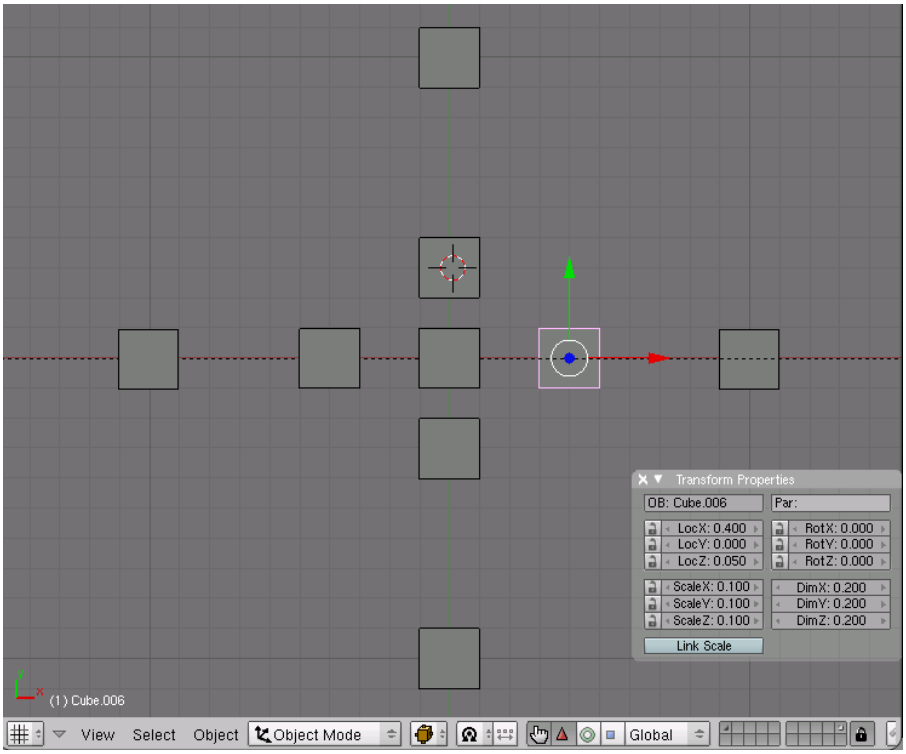

Figure 43: Blender scene – cubes moved from 1 to 2 meters away in the range direction

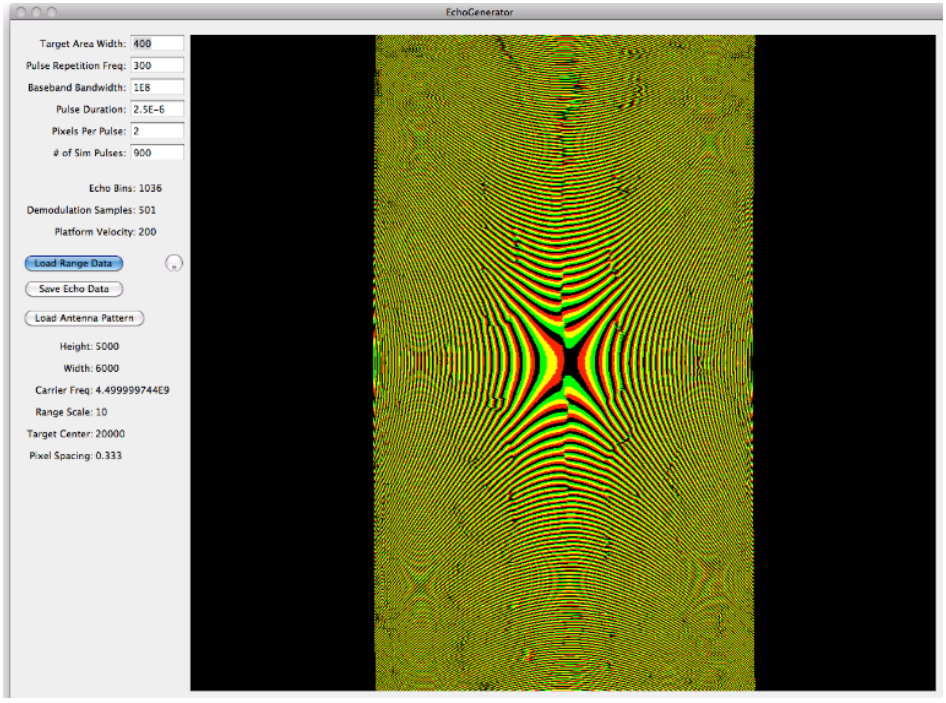

Figure 44: Echo generated – cubes moved from 1 to 2 meters away in the range direction

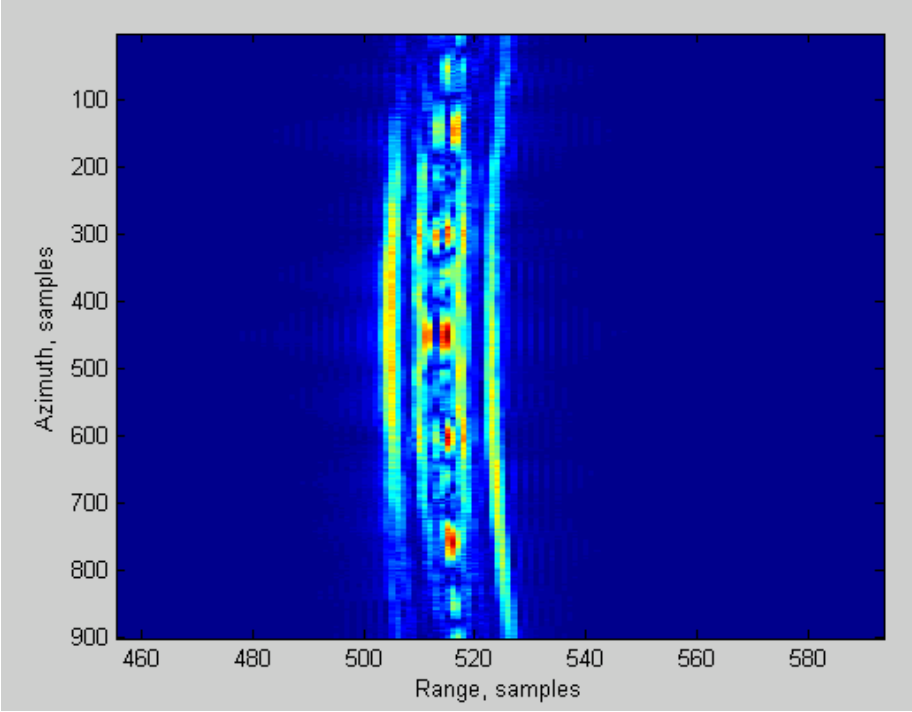

Figure 45: Range compressed SAR image – cubes moved from 1 to 2 meters away in the range direction

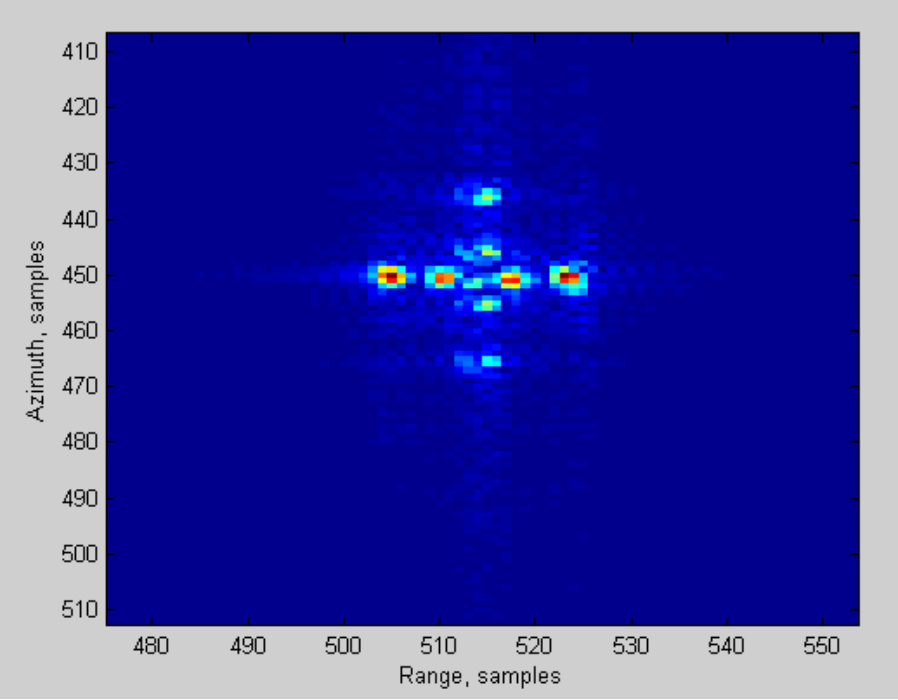

Figure 46: Final SAR image – cubes moved from 1 to 2 meters away in the range direction

The final SAR image in Figure 46 shows that placing 2 meters separation between the cubes in the range direction reduced the ambiguity observed in Figure 42. However, the fact that the center cube is still not visible in Figure 46 indicates that the ambiguity in the range direction was not completely removed, therefore further experimentations were made.

A third simulation was run with the cubes 2 meters away from the center cube in both the range and azimuth directions (see Figure 47). The echo generated, the SAR image after azimuth compression and the final SAR image are shown in the Figures 48, 49 and 50.

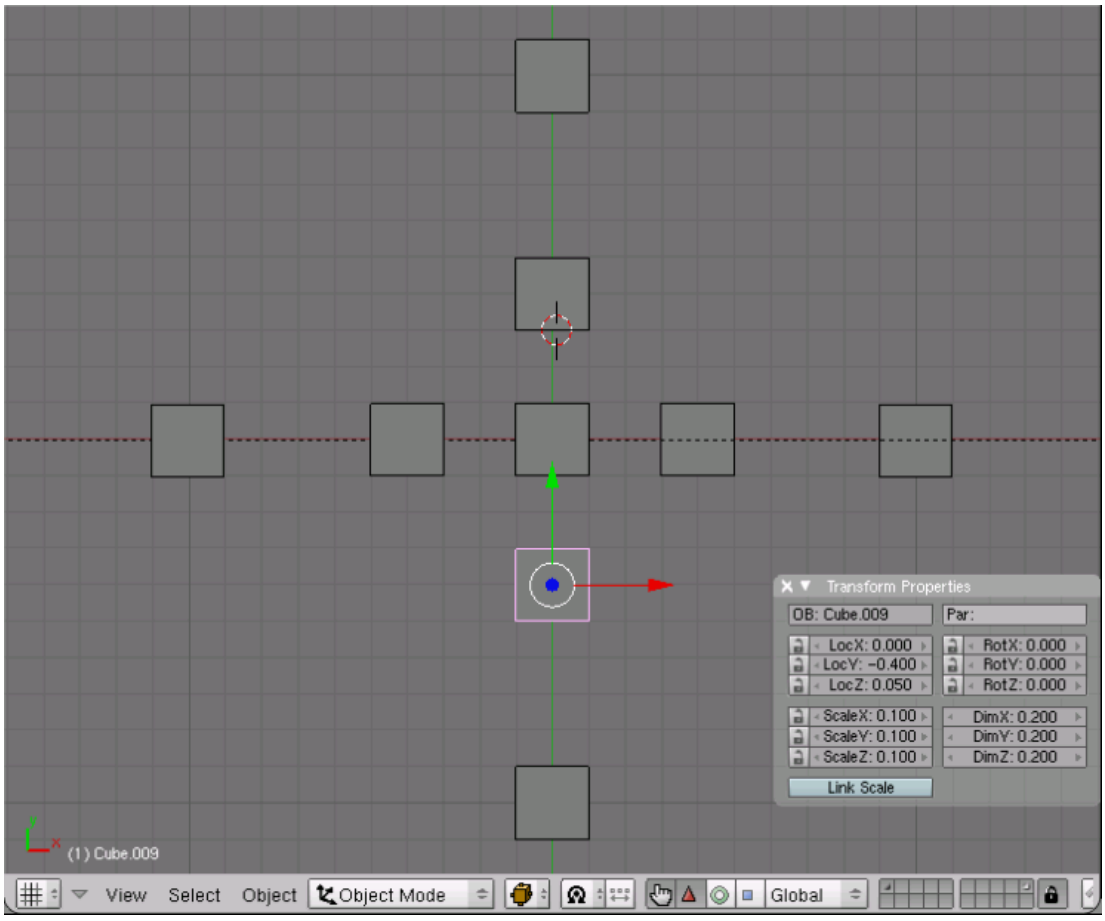

Figure 47: Blender scene – cubes 2 and 8 meters away in both directions

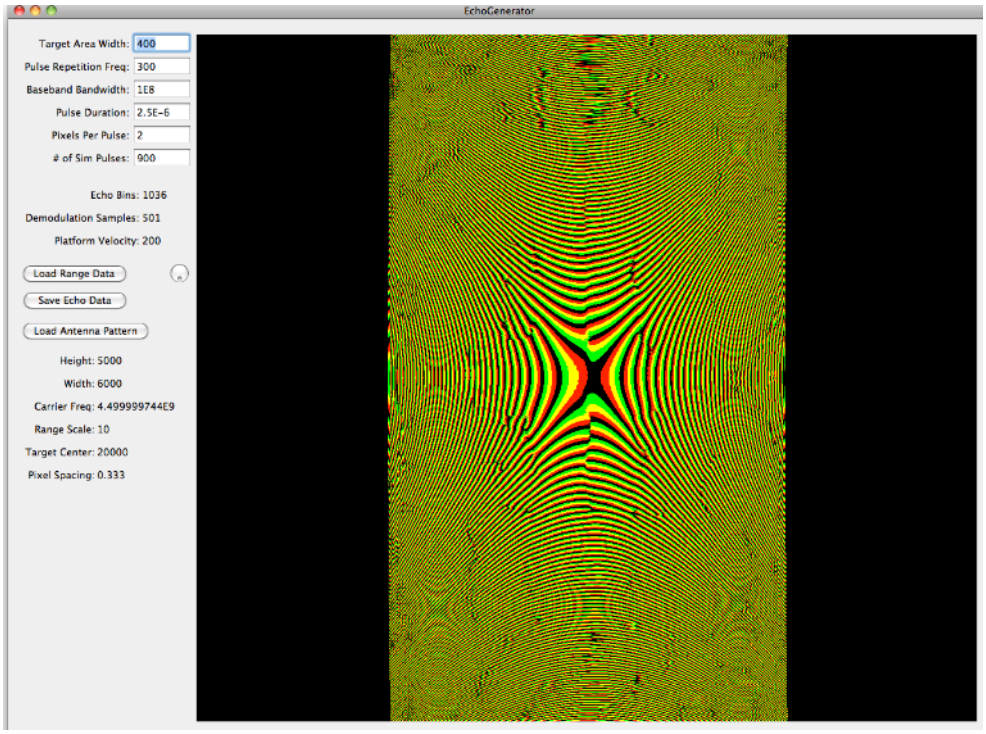

Figure 48: Echo generated – cubes 2 and 8 meters away in both directions

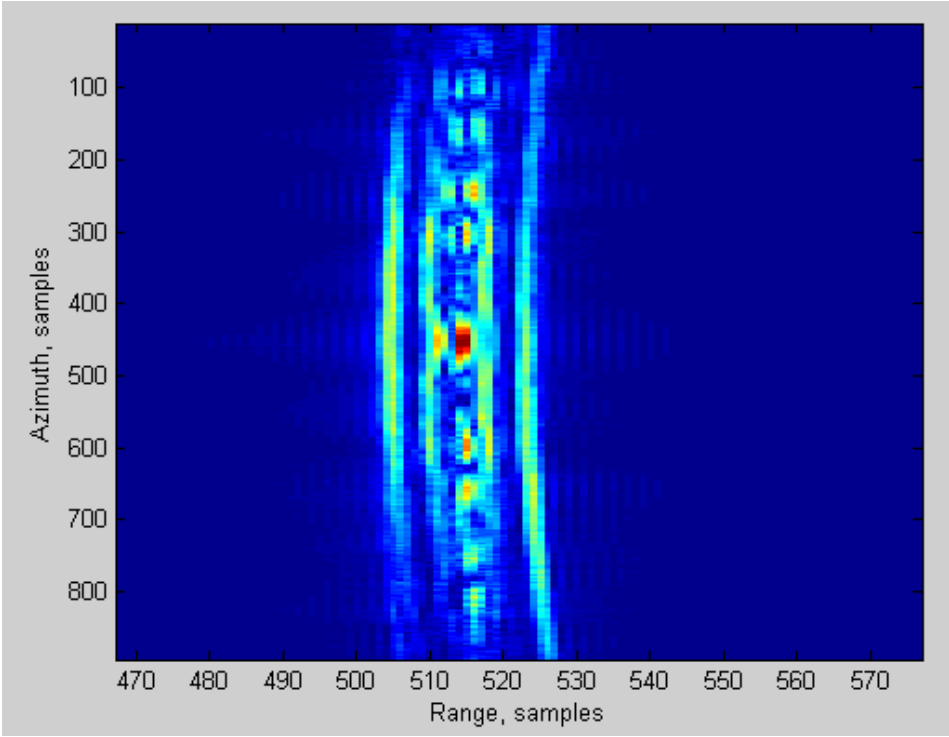

Figure 49: Range compressed SAR image – cubes 2 and 8 meters away in both directions

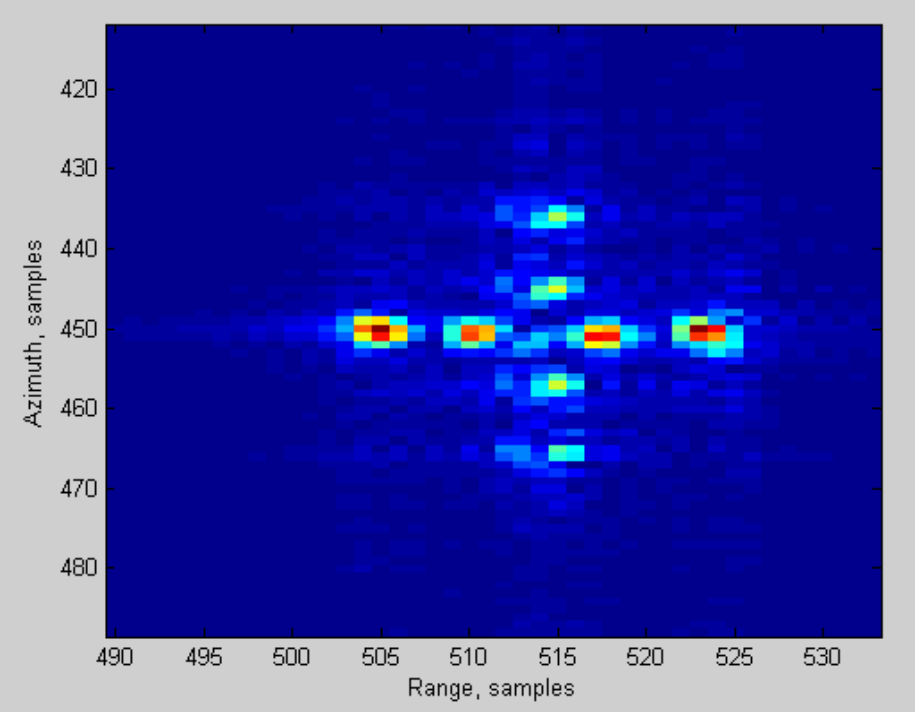

Figure 50: Final SAR image – cubes 2 and 8 meters away in both directions

The final SAR image in Figure 50 clearly shows all of the cubes that are 2 and 8 meters away from the center in both the azimuth and range directions, but the center cube is still not visible.

A fourth simulation was run in which the cubes that were 2 meters away from the center cube in the range direction were moved an extra meter away (see Figures 51 and 52). The echo generated, the SAR image after azimuth compression and the final SAR image are shown in the Figures 53, 54 and 55.

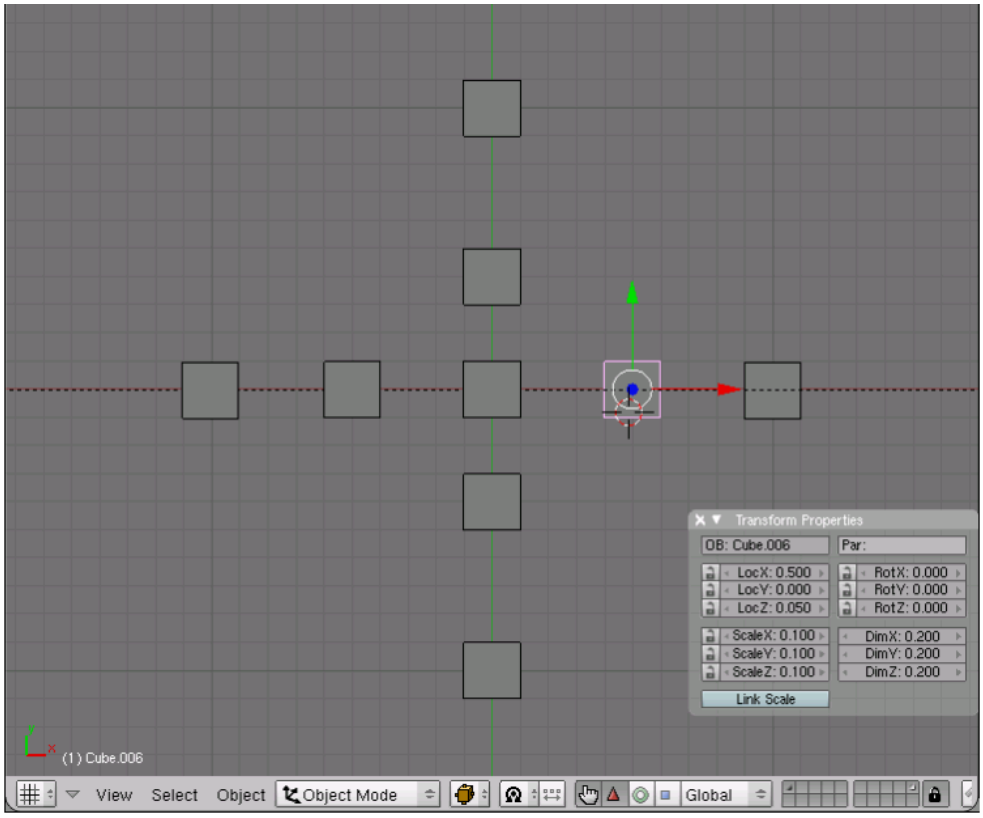

Figure 51: Blender scene – cubes moved from 2 to 3 meters away in the range direction

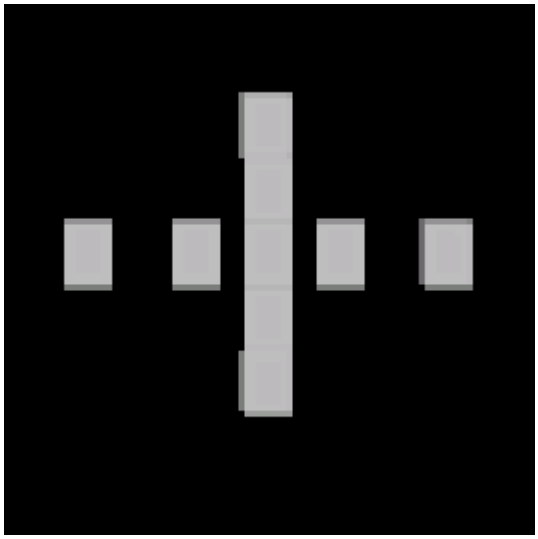

Figure 52: Rendered Blender image – cubes moved from 2 to 3 meters away in the range direction

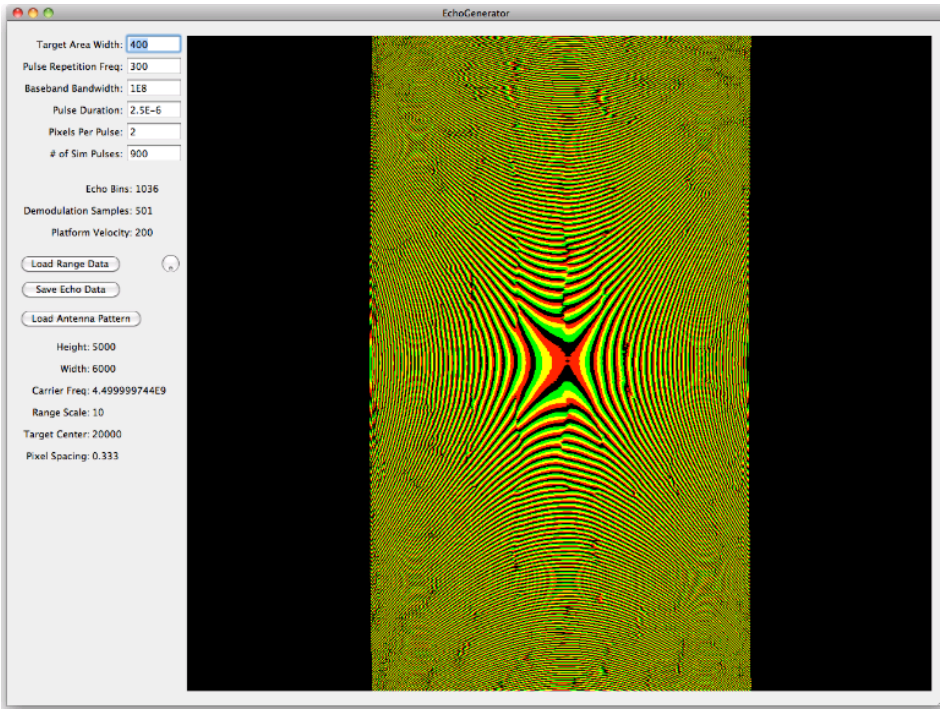

Figure 53: Echo generated – cubes moved from 2 to 3 meters away in the range direction

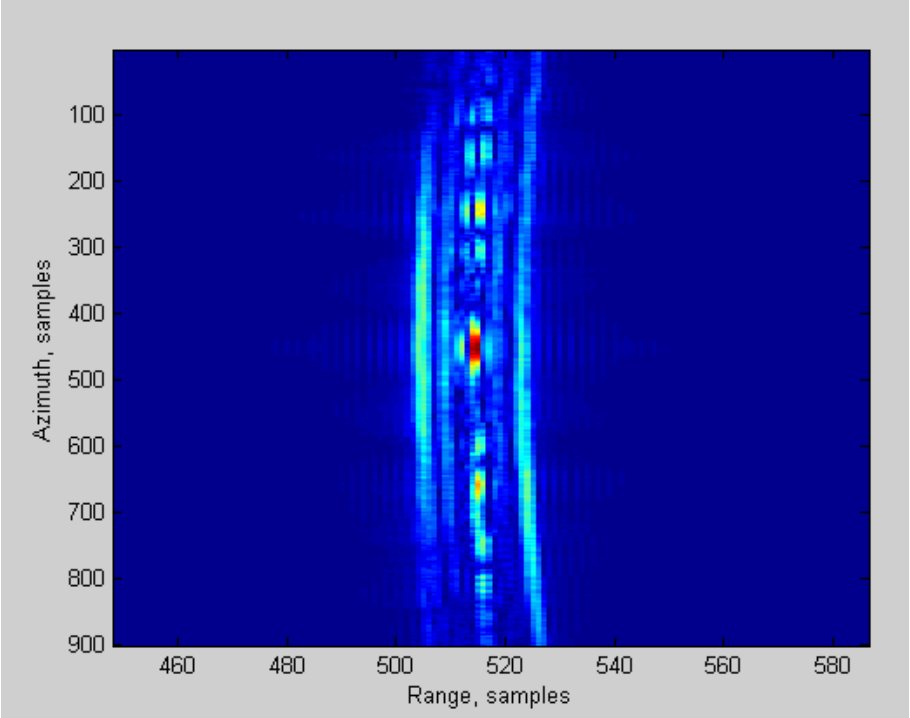

Figure 54: Azimuth compressed SAR image – cubes moved from 2 to 3 meters away in the range direction

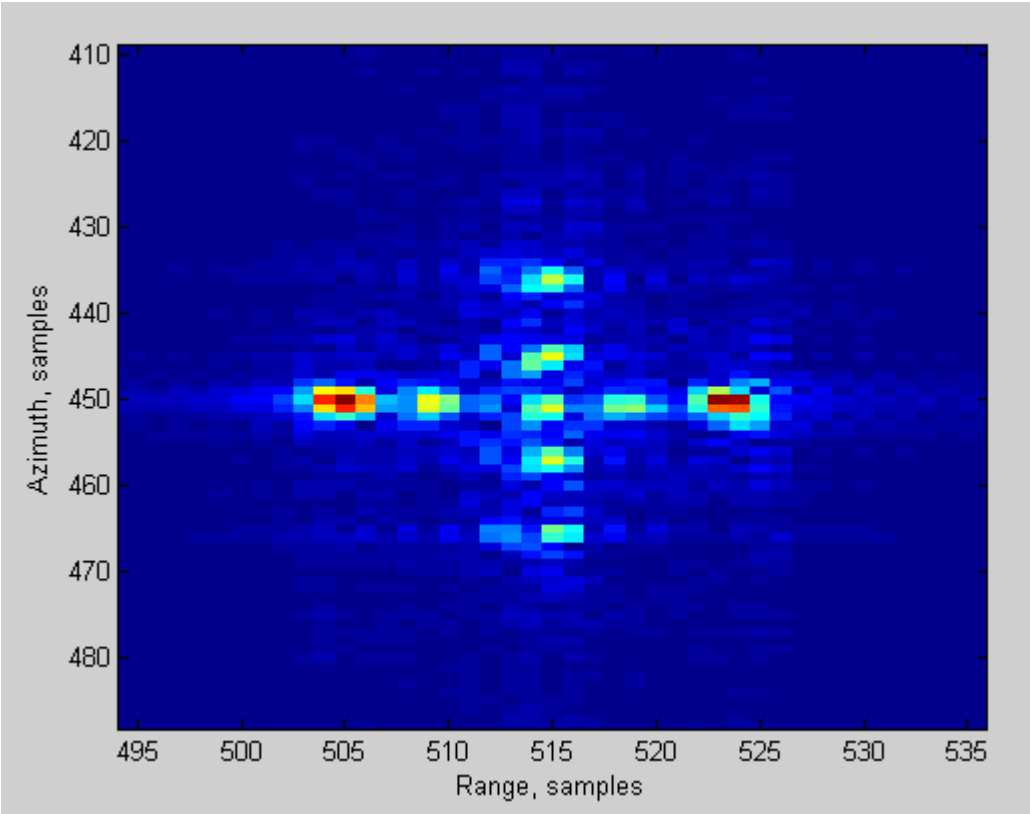

Figure 55: Final SAR image – cubes moved from 2 to 3 meters away in the range direction

The final SAR image above shows that when the spacing between the center cube and the cubes in the range direction is increased from 2 to 3 meters, the center cube is much more recognizable.

For completeness, a further simulation was run in which the cubes that were 2 meters away from the center cube in the azimuth direction were moved an extra meter away, as shown in Figures 56 and 57. The echo generated, the SAR image after azimuth compression and the final SAR image are shown in the Figure 58, 59 and 60.

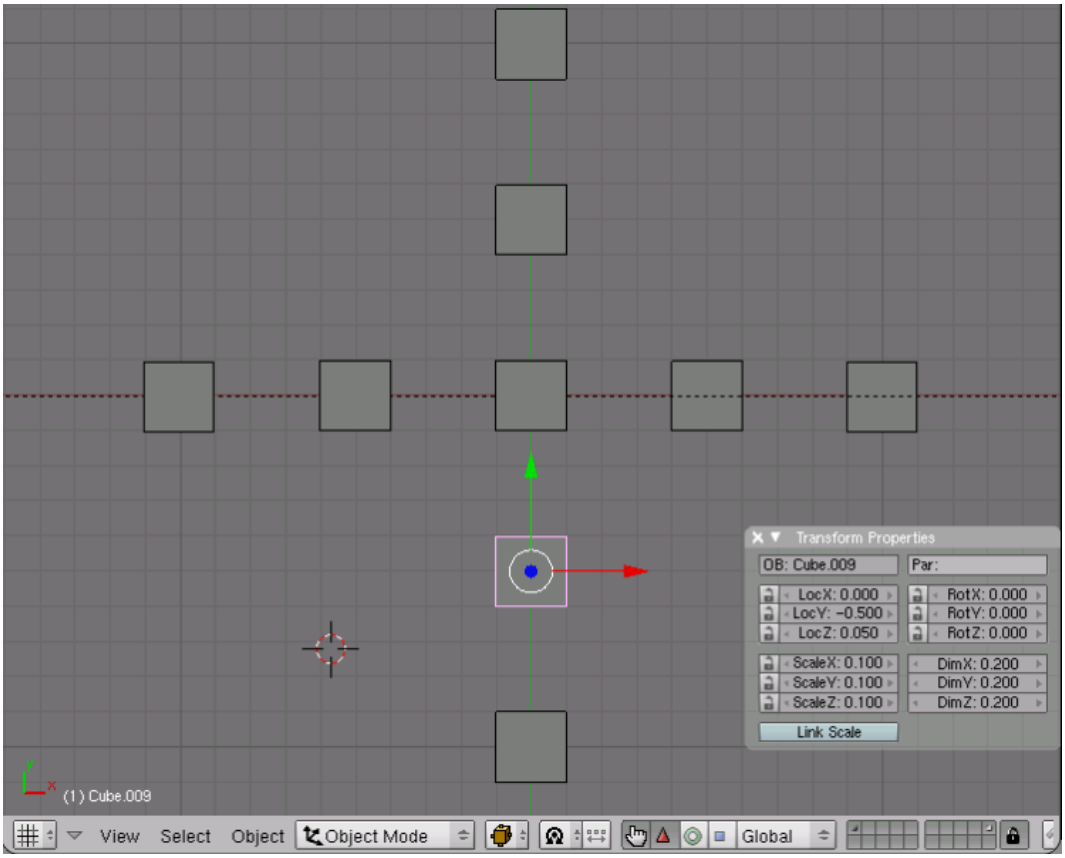

Figure 56: Blender scene – cubes 3 and 8 meters away in both directions

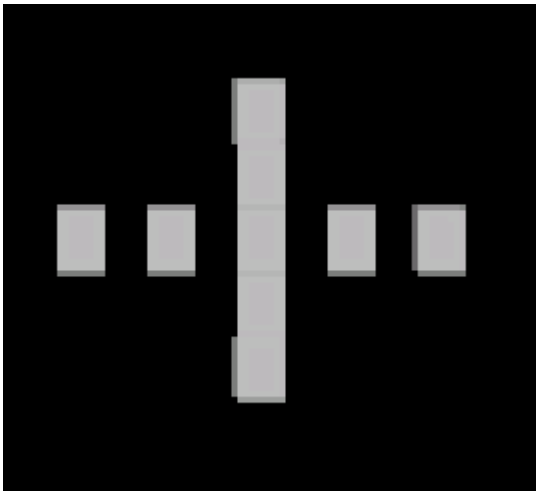

Figure 57: Rendered Blender image – cubes 3 and 8 meters away in both directions

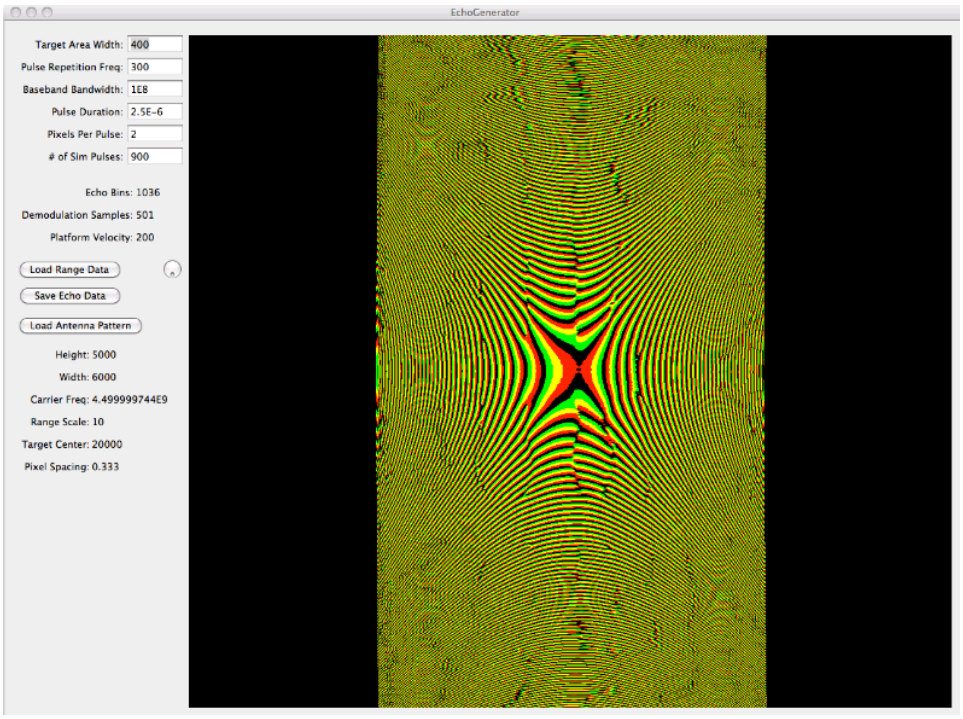

Figure 58: Echo generated – cubes 3 and 8 meters away in both directions

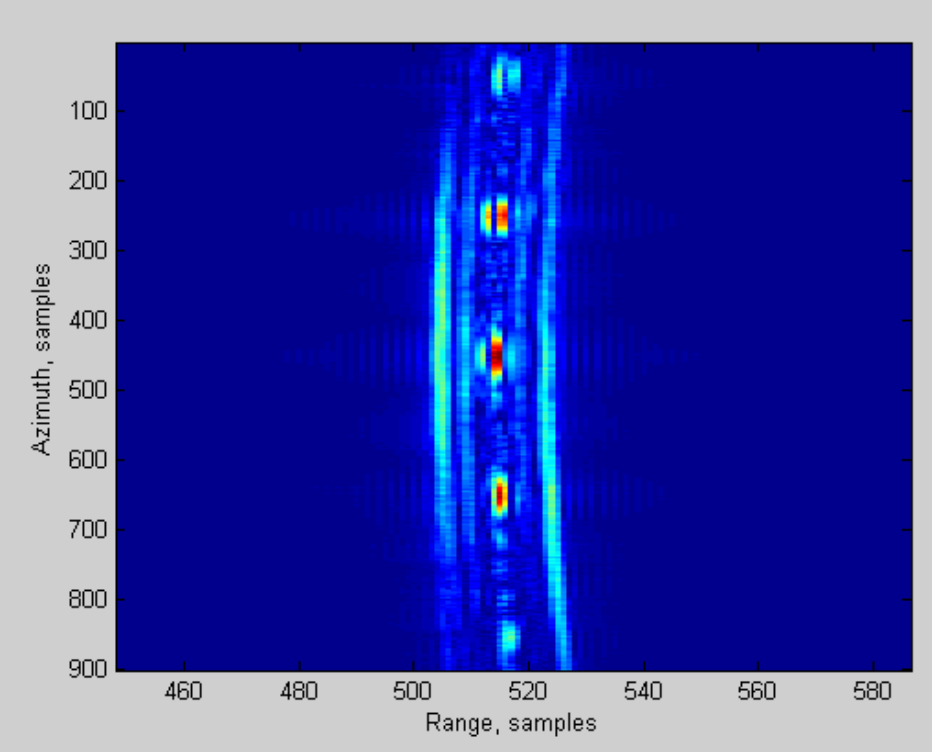

Figure 59: Range compressed SAR image – cubes 3 and 8 meters away in both directions

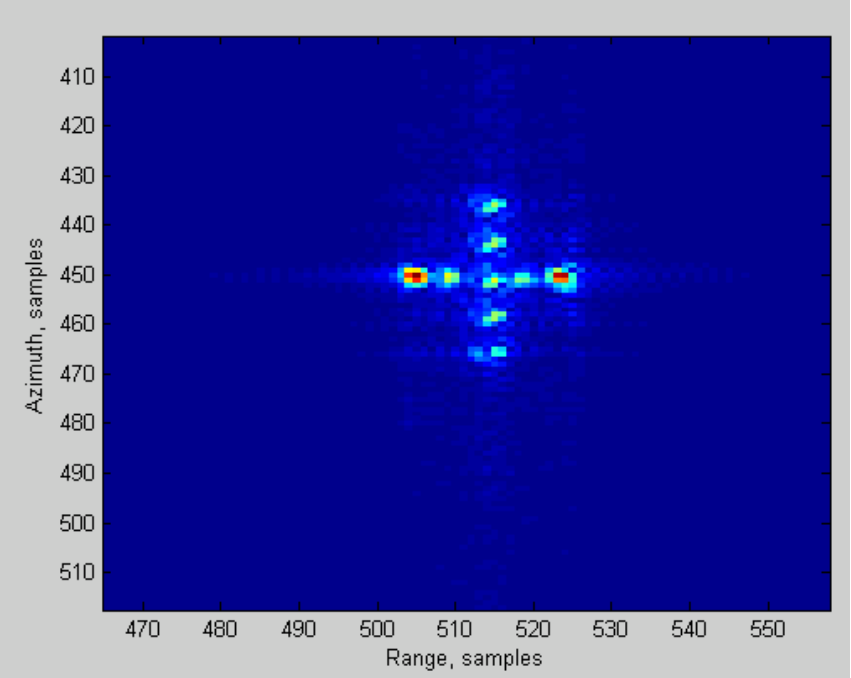

Figure 60: Final SAR image – cubes 3 and 8 meters away in both directions

The final SAR image above shows all the cubes in the both the range and azimuth direction clearly defined and visible.

In summary, using an elevation angle of 45 degrees in the above simulations resulted in accurate simulations to a range resolution of 3 meters and an azimuth resolution of 1 meter. The experiments proved that the range resolution is a more sensitive variable than the azimuth resolution.

6.2. Tank experiments

In order to test the ability of simulation to detect target features, simulations were run on two tanks modeled and rendered in blender. These tanks vary in size, material, and shape. The material reflectance value added to the tanks was 0.5 to the body, 0.8 to the turret, and 1 to the barrel. The first, a 2s-1 tank is shown in Figure 61 alongside a Blender scene reflecting its key

body dimensions. The echo generated, the SAR image after azimuth compression and the final SAR image are shown in the Figures 62, 63 and 64.

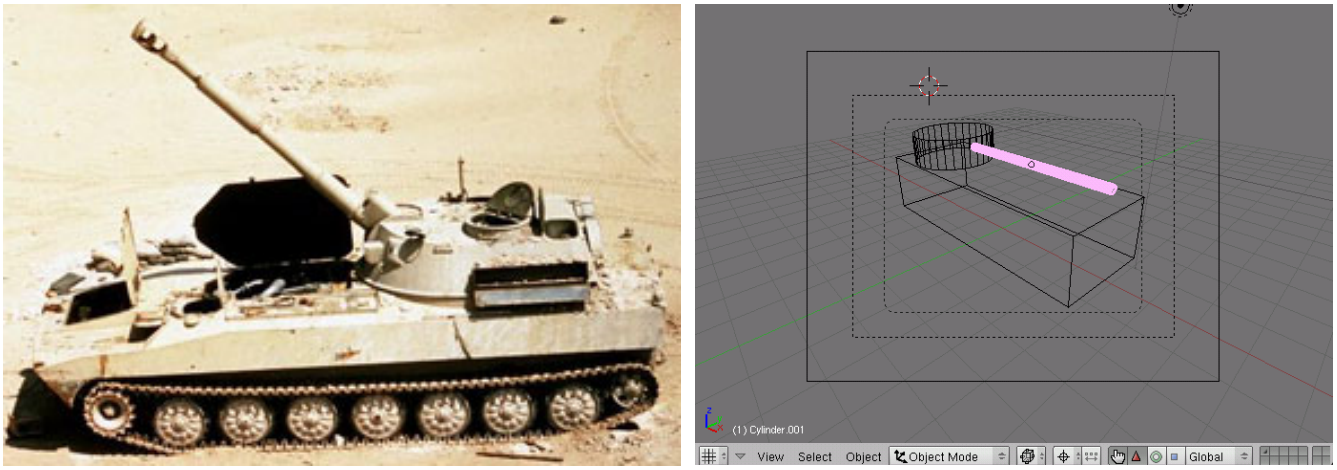

Figure 61: Image[3] and Blender scene – 2s-1 tank

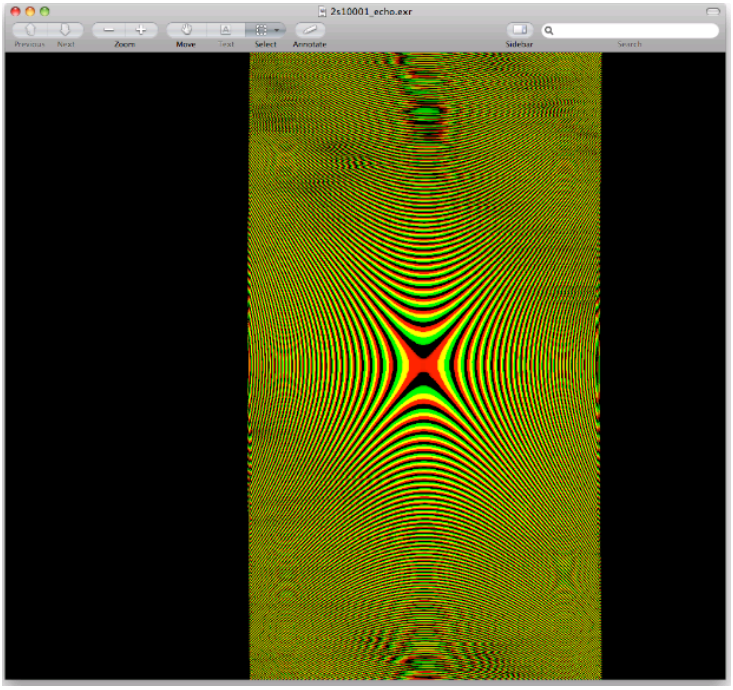

Figure 62: Echo generated – 2s-1 tank

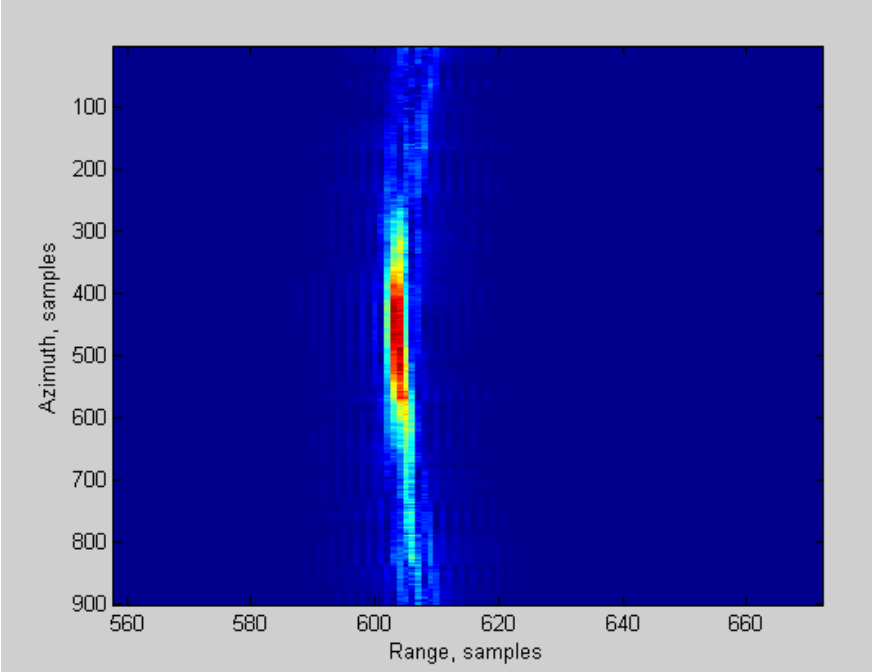

Figure 63: Range compressed SAR image – 2s-1 tank

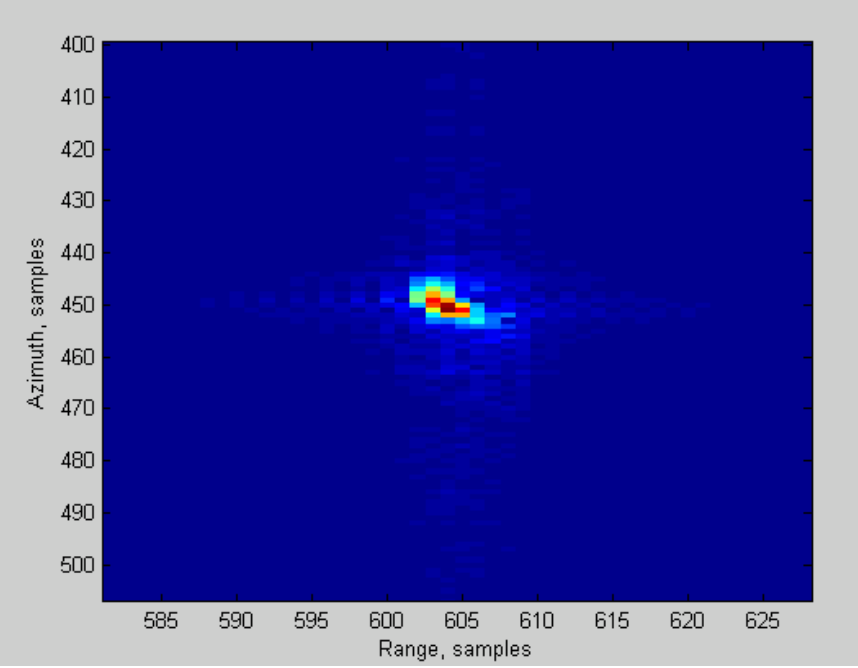

Figure 64: Final SAR image – 2s-1 tank

A T62 tank (Figure 65) was also modeled and rendered in Blender as shown in Figures 66 and 67. The echo generated, the SAR image after azimuth compression and the final SAR image are shown in the Figures 68, 69 and 70.

Figure 65: T62 tank[3]

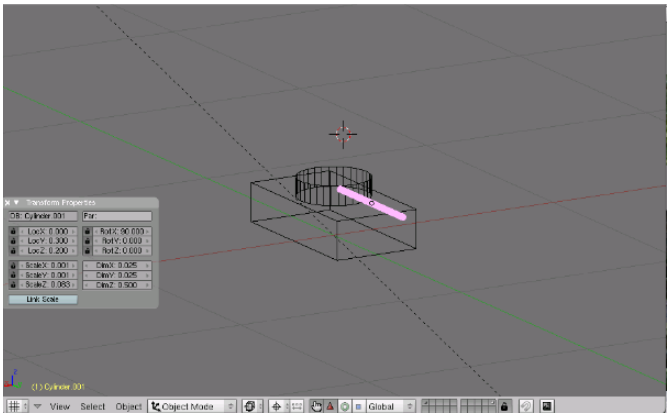

Figure 66: Blender scene – T62 tank

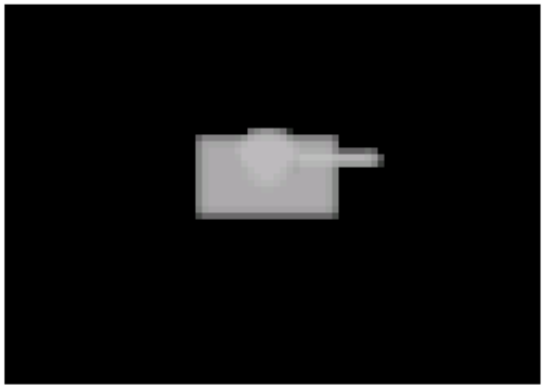

Figure 67: Rendered Blender image – T62 tank

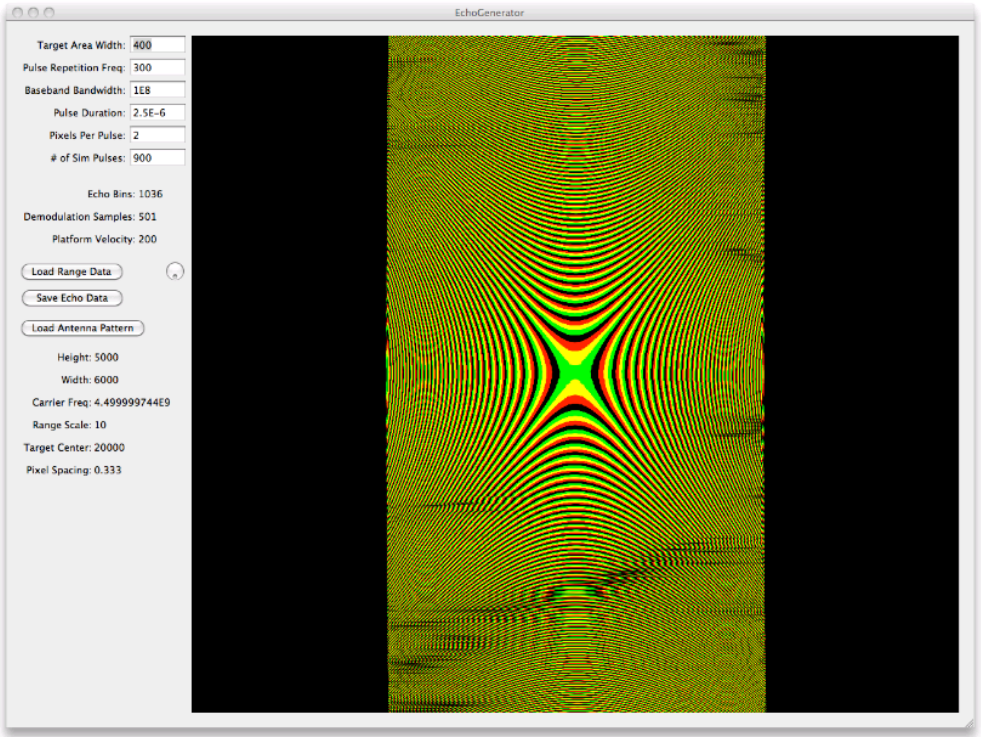

Figure 68: Echo generated – T62 tank

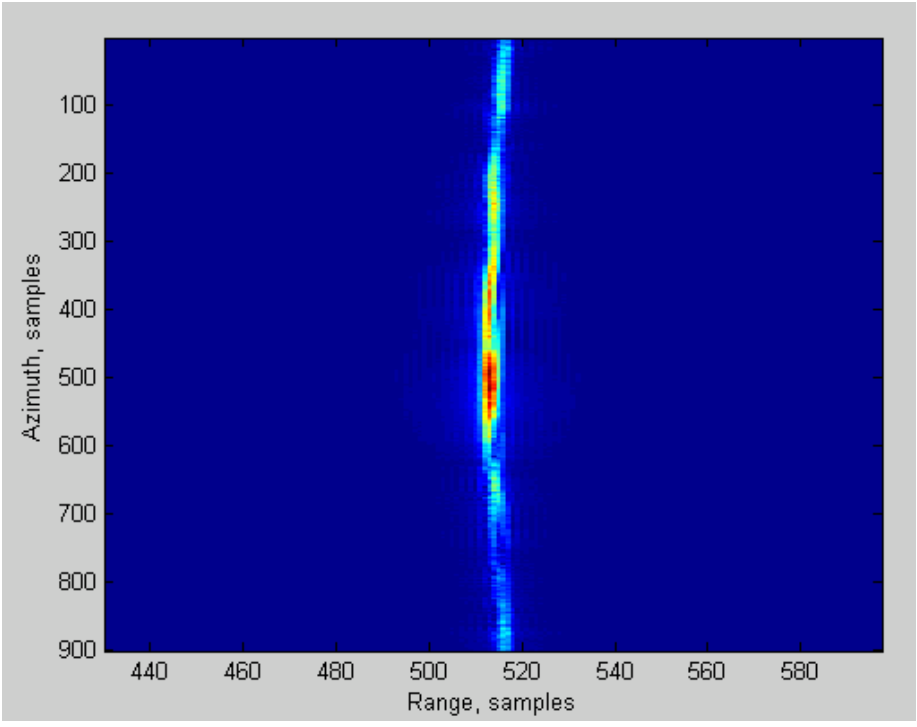

Figure 69: Range compressed SAR image – T62 tank

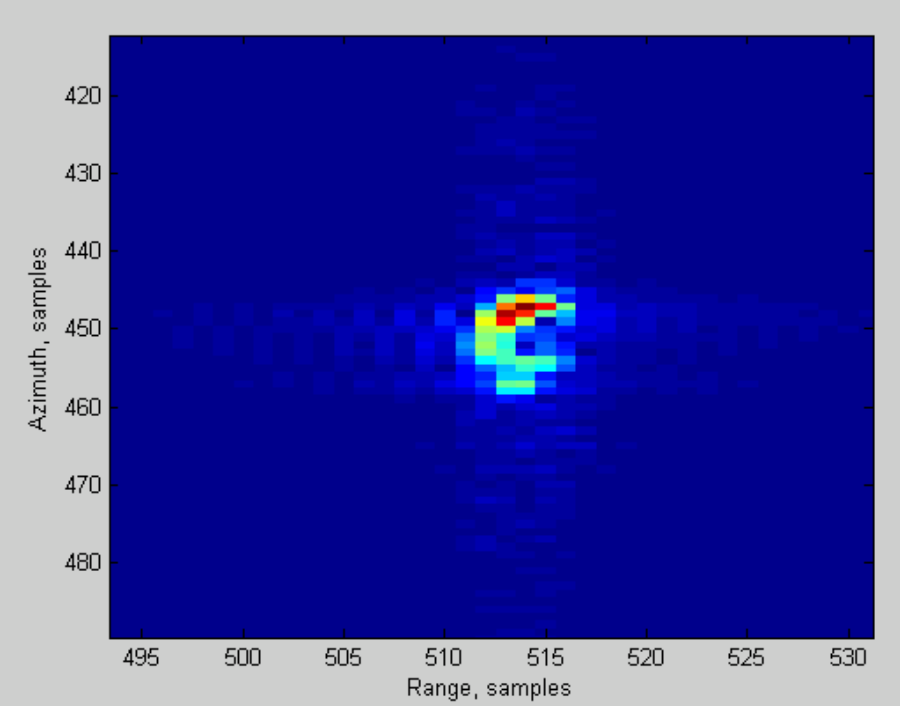

Figure 70: Final SAR image – 2s-1 tank

The results in Figures 64 and 70 show that the simulation was able to detect each of the tanks. Also, the images show how the return from the body of the tanks was moderate, compared to the high return from the barrels.

7. VALIDATION

Having obtained the encouraging results set out in Section 6 (above), the next step was to attempt to validate the simulation results by comparing them to real SAR data. However, as explained earlier in this paper, most SAR images used for military ATR are classified and not available to the general public, and the only useful data available is a set of MSTAR images.

The MSTAR images were taken in spotlight mode, whereby the angle of the illumination on the desired location changed across the course of the platform's flight path. However, as explained in sections 1.4 and 5 above, this thesis used stripmap mode for all simulations and the angle of illumination was fixed at 90 degrees. There is a set of MSTAR images of a 2s-1 tank taken at a 45 degree elevation angle. In each of the MSTAR images, the tank was rotated by 5 degrees from 0 to 90 degrees. A matching simulation was performed by rotating the blender scene set out in Figure 61 above. This enabled the results to be compared to the MSTAR images at each angle (see Figures 71 to 89).

However, in addition to the differing SAR modes of operation (spotlight and stripmap), there are other differences in the parameters used by MSTAR compared to those used by the simulation in this thesis. Firstly, MSTAR data has a resolution of 1 foot compared to the 1 meter resolution in this simulation. Secondly, the tank models in this thesis were approximated by using squared figures and approximating its reflection strength. Third, no background imagery was added around the models rendered in blender and therefore the final SAR image doesn’t show ground reflections that the MSTAR images show around the targets. Finally, noise wasn’t added to the simulation. All of these factors compromise the accuracy of the comparison between the final SAR simulations and the MSTAR images shown below.

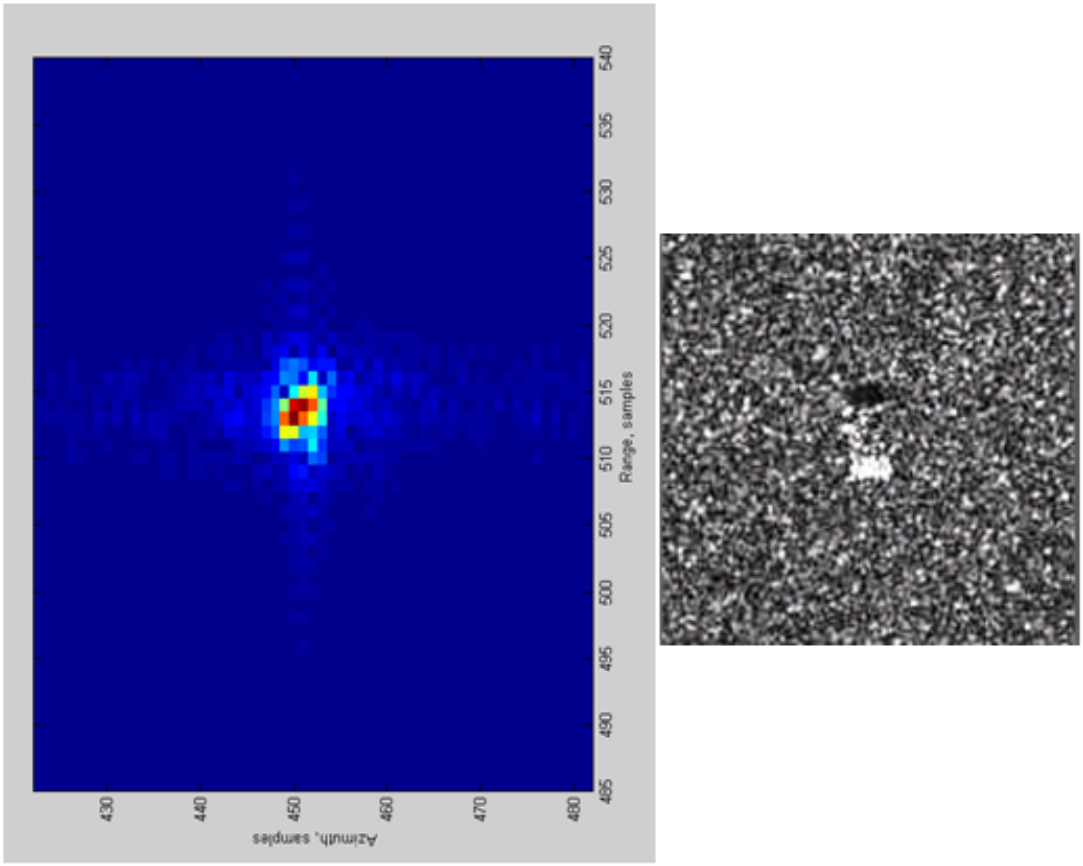

Figure 71: Final SAR image (left) and MSTAR image (right) – 0 rotation

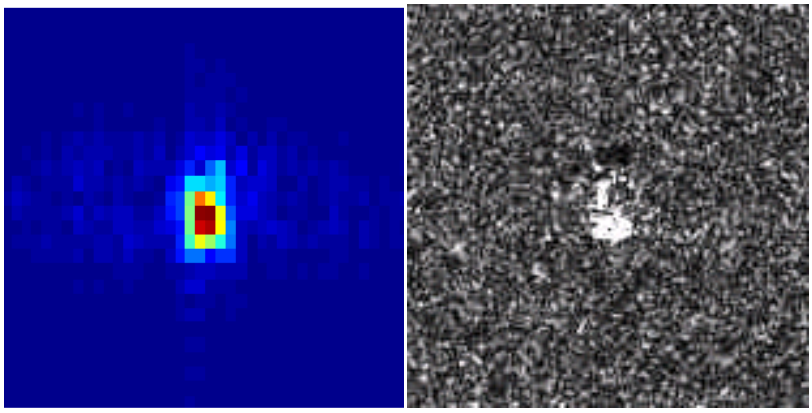

Figure 72: Final SAR image (left) and MSTAR image (right) – rotated 5°

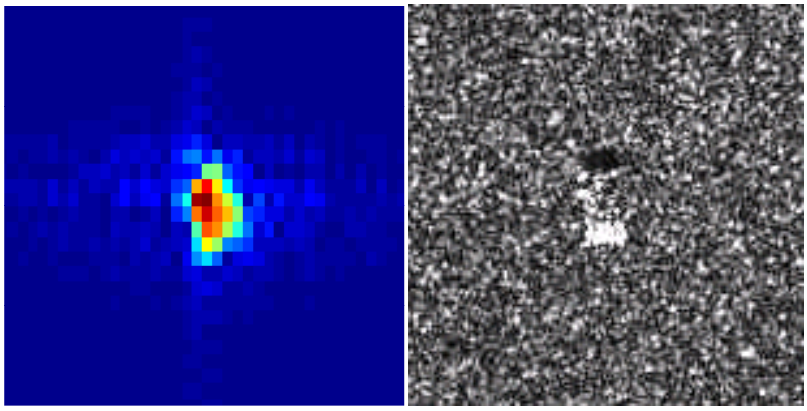

Figure 73: Final SAR image (left) and MSTAR image (right) – rotated 10°

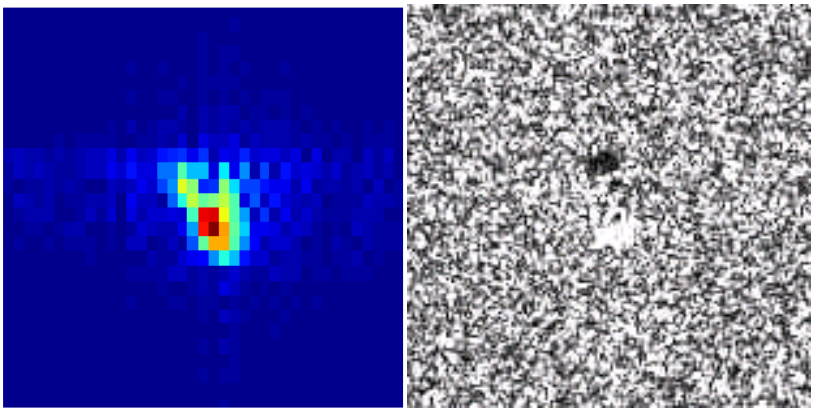

Figure 74: Final SAR image (left) and MSTAR image (right) – rotated 15°

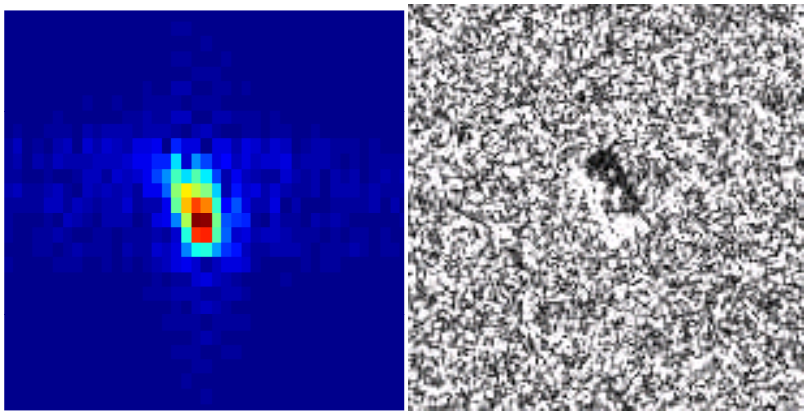

Figure 75: Final SAR image (left) and MSTAR image (right) – rotated 20°

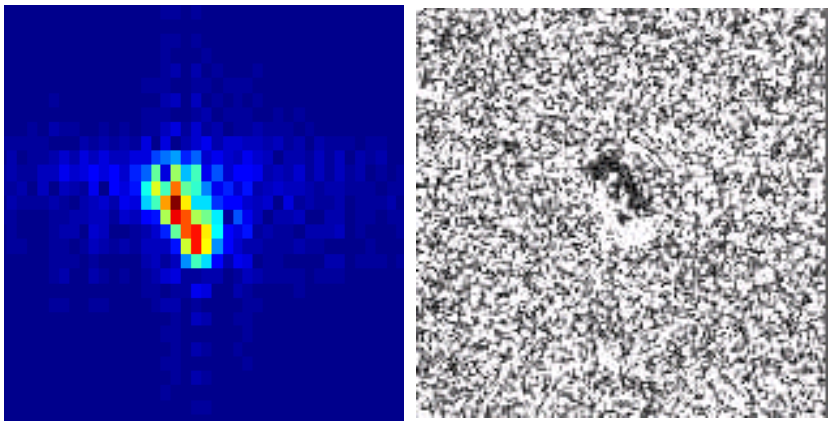

Figure 76: Final SAR image (left) and MSTAR image (right) – rotated 25°

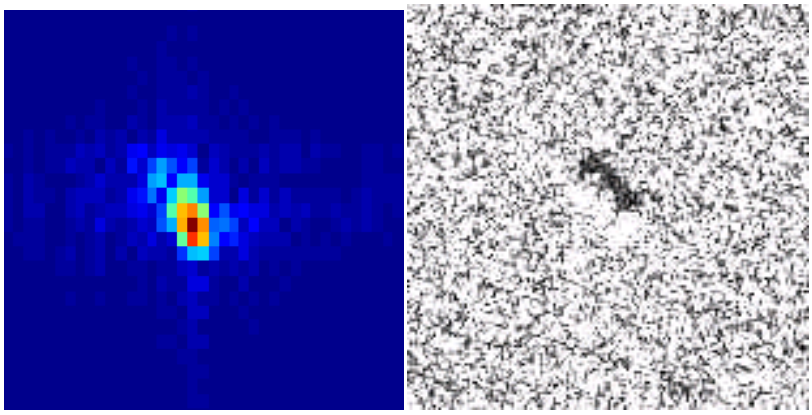

Figure 77: Final SAR image (left) and MSTAR image (right) – rotated 30°

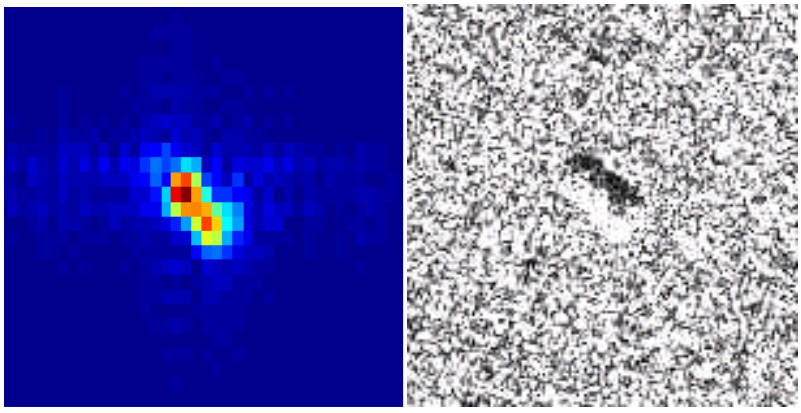

Figure 78: Final SAR image (left) and MSTAR image (right) – rotated 35°

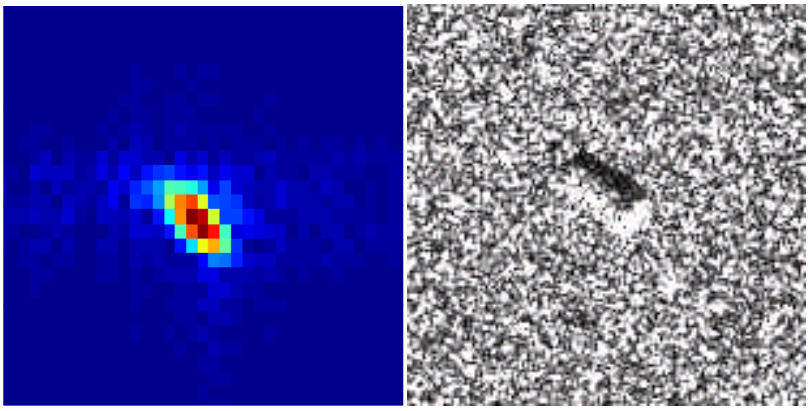

Figure 79: Final SAR image (left) and MSTAR image (right) – rotated 40°

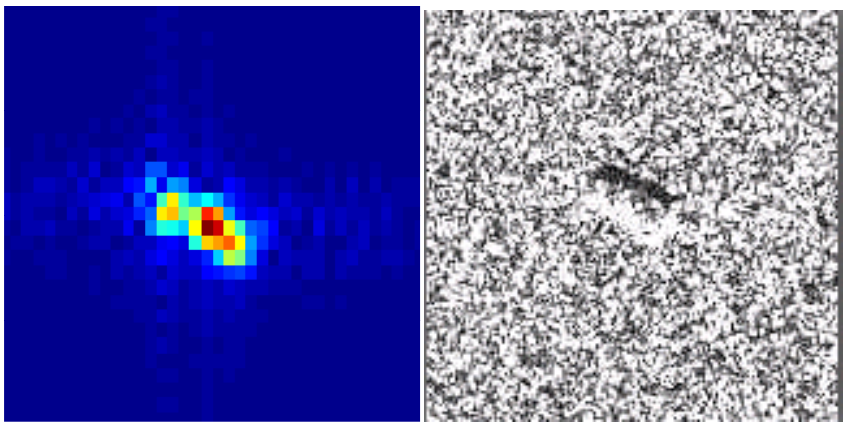

Figure 80: Final SAR image (left) and MSTAR image (right) – rotated 45°

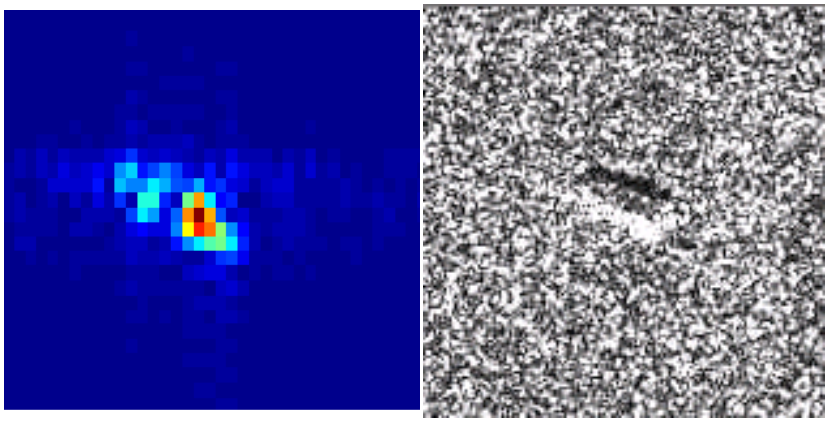

Figure 81: Final SAR image (left) and MSTAR image (right) – rotated 50°

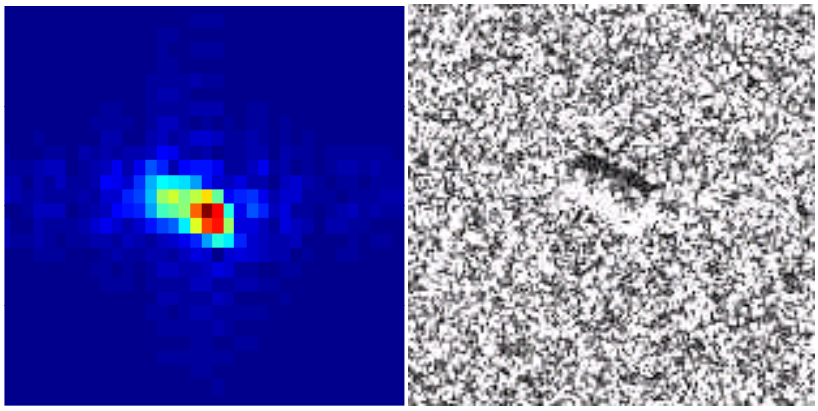

Figure 82: Final SAR image (left) and MSTAR image (right) – rotated 55°

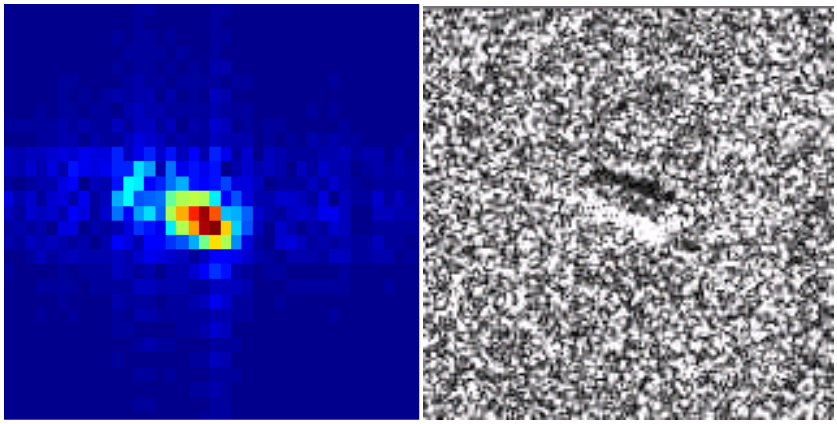

Figure 83: Final SAR image (left) and MSTAR image (right) – rotated 60°

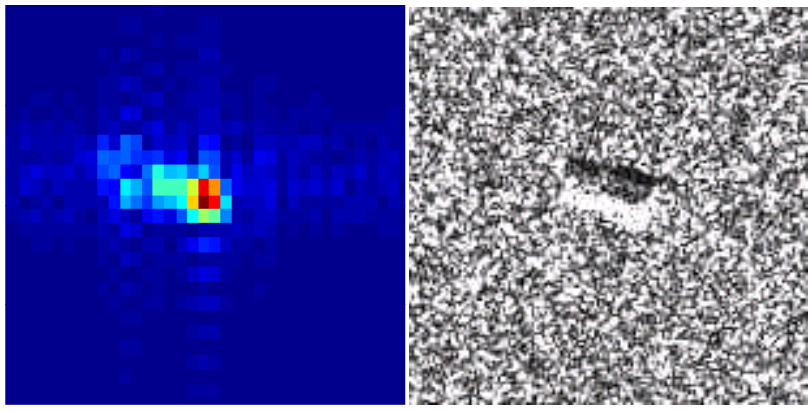

Figure 84: Final SAR image (left) and MSTAR image (right) – rotated 65°

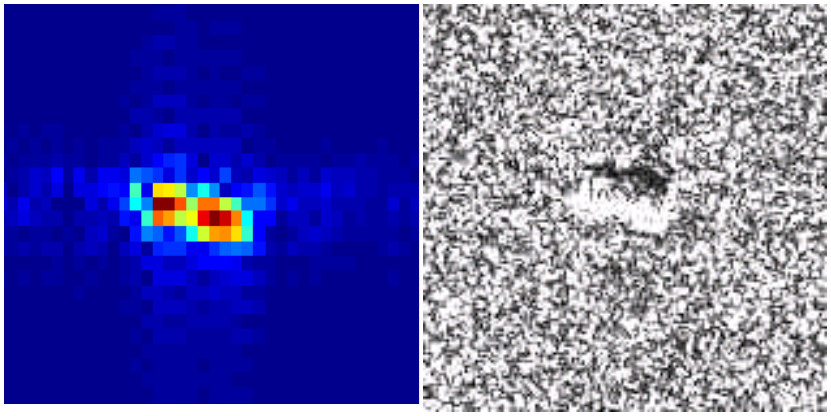

Figure 85: Final SAR image (left) and MSTAR image (right) – rotated 70°

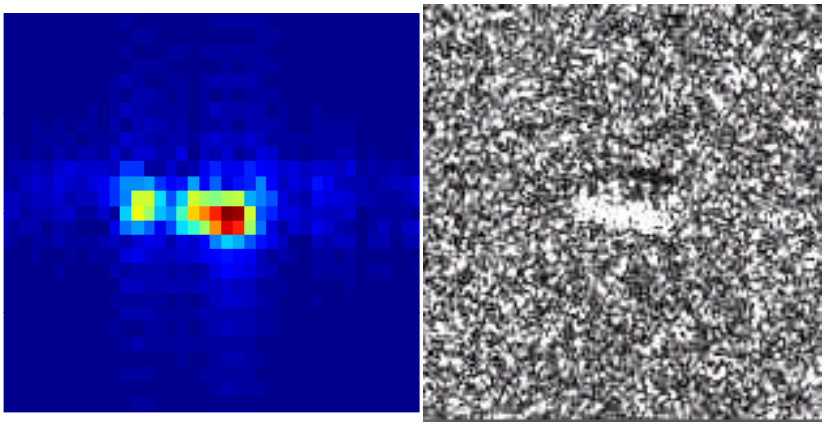

Figure 86: Final SAR image (left) and MSTAR image (right) – rotated 75°

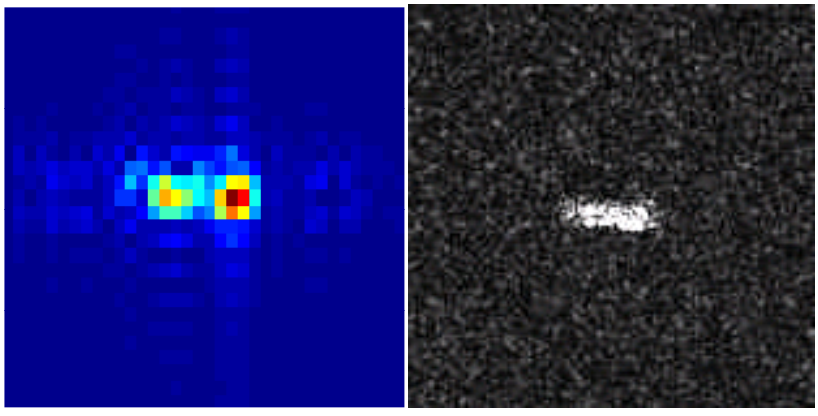

Figure 87: Final SAR image (left) and MSTAR image (right) – rotated 80°

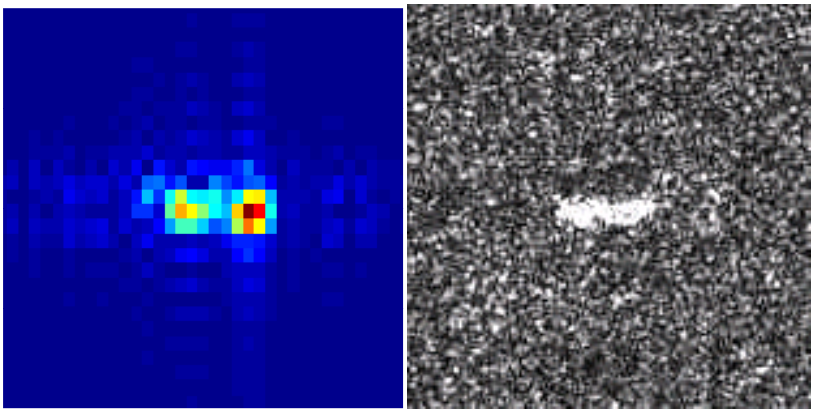

Figure 88: Final SAR image (left) and MSTAR image (right) – rotated 85°

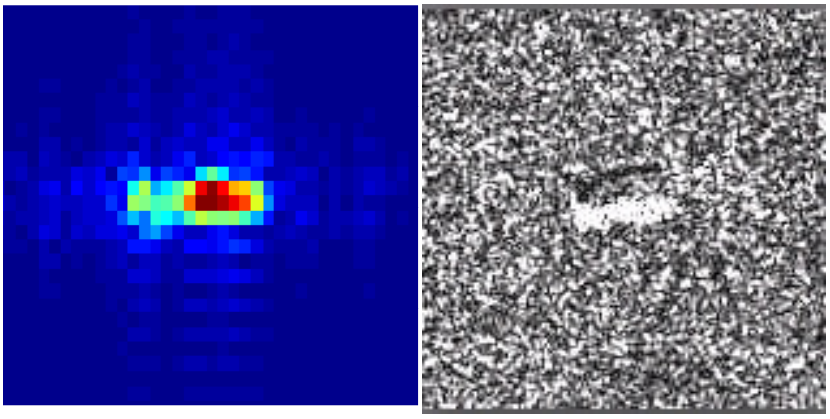

Figure 89: Final SAR image (left) and MSTAR image (right) – rotated 90°

As can be seen from Figure 71 (above), when the 2s-1 tank is rotated 0 degrees, a lot of reflection is seen in the front on the tank and there is a slight shadow towards the back of the tank where the turret is located, which correlates with the matching MSTAR image. As the 2s-1 tank model is rotated from 0 to 50 degrees, the front of the tank gradually reflects less and the reflection from the side of the tank becomes stronger. Once the 2s-1 tank model is at 50 degrees, the entire side of the tank is reflecting but the definite square shape of the side of the tank is still not too clear. When the tank is rotated by 80 degrees (as shown in Figure 87), the square side of the tank is completely visible and correlates well with the matching MSTAR image. From 80 to

90 degrees the square shape of the side of the tank becomes more defined and the reflection becomes more even across the entire side of the tank.

As can be seen from Figures 71 to 89, despite the incongruities between the simulation parameters used in this thesis and those used to obtain the MSTAR images, there is a high degree of similarity between the results at each angle.

8. CONCLUSION

During the course of this thesis project, a more realistic SAR simulation was developed which, for the first time in a Cal Poly simulation, did not hardcode reflectivity or range data. This represents a very important step towards creating a realistic simulator that can be used in ATR testing and development. An optimized implementation of the simulation was also obtained by porting the antenna pattern and echo generation to C++.

The antenna radiation pattern and the quadrature demodulation signature were also optimized by only calculating them once, resulting in a notable reduction in processing time for each simulation. The importance of this optimization should not be understated, particularly in relation to the processing of high resolution images.

Initially, when the method of rendering individual images in Blender was used to obtain the perspective of the radar platform as it flies by, a variance in the reflectance and range obtained from Blender resulted in bad imagery. Therefore, the method of rendering one large image and processing sub-frames of that large image was used, with the advantage that it processes more quickly but the disadvantage that it cannot process images from the different perspectives of the antenna platform. The major drawback of not running the simulation in perspective mode is that certain point targets do not appear and disappear during the platform motion, as they would in real SAR.

By processing sub-frames of a large image, the simulation can run in about the same processing time as in previous simulations. However, the image being processed has a resolution that is several orders of magnitude greater than the images processed in prior simulations. Using the simulator developed in this thesis, a 7800 by 5000 pixel image can be processed in less than an

hour, which would have taken a few days to process using the previous simulator. Processing a large image means that several relatively small targets can be detected within a large target plane. The capacity to image more detailed targets is a very important step towards developing a simulation that can be used for ATR.

In other words, a simulation has been developed that can detect higher resolution targets which can be placed in a synthetic background. Previous simulations only dealt with small point targets and a limited number of reflectors within the image, and therefore could not create data that could be run through ATR algorithms. The enhancement of being able to use Blender to generate complex scenes enables future work on algorithms that can determine if something is a target or a clutter, and generates images that can be useful for the actual ATR algorithms themselves. In addition, the simulation is running in less than 1 hour, meaning that it is possible to create more data in less time rather than being limited to a small number of runs because the simulator takes days to complete.

8.1. Future work

There are several areas of consideration for future work in this project. The method of being able to obtain the perspective of the radar platform as it flies by taking multiple images gave suboptimal results, and because of time constraints an alternate method was used. By using multiple images, different modes of SAR operation could be simulated. Research can be done on the use of rendering individual images in Blender to create final SAR data. By rendering individual images, the issue of not running the simulation in perspective mode would be mitigated and therefore the return from faces of objects with a high angle of incidence would be visible. Changing the camera path in Blender could show the effects of the platform motion not

being uniform, and could simulate different modes of SAR (such as spotlight and scan SAR modes).

Different antenna radiation patterns could also be used for experimentation, such as parabolic radar antennas and circular radar antennas. The parameters used in the antenna pattern and echo generation sequence GUI can be adjusted in order to find an optimized simulation for a given situation.

Reflection properties could also be added to the targets created in Blender, for more realistic echo generation. Manipulating different material properties in Blender, such as reflectivity, emittance and scattering, can all be used to create a more realistic return pattern.

In the future, a comparison could be made of different SAR image generation algorithms, such as Chirp Scaling, Omega-K and SPECAN. These algorithms can be either implemented in Matlab or in another programming language for better performance and integration with the other modules (antenna radiation pattern and echo generator).

Current and past Master's theses at Cal Poly have been restricted to a data set of Moving and Stationary Target Acquisition and Recognition Radar (MSTAR) that were collected in spotlight mode and do not correlate with what is being simulated (in stripmap mode). It would be useful if stripmap SAR data could be obtained in order to compare the results generated from the simulation with real SAR images.

Future work could be done to modify the RCMC application so that RCMC is applied for the current range of bins, and not just the center bin as in the current simulation. Using different

center points within the RCMC calculation would allow targets that are away from center (450th azimuth bin) to be imaged more accurately.

This simulation has the antenna pattern set up to be 1 degree greater than the 3dB width of the azimuth beam. The size of the image could be increased in order to bring in more peripheral reflections. The alternative is to increase the pixel separation and keep the current image size, resulting in decreased feature resolution.

The antenna pattern could also be set up to be larger than the image and sweep across the smaller image. This would cause everything within the image to always reflect given that it is covered by the antenna pattern at all times.

List of references

1. Soumekh, Mehrdad. *Synthetic Aperture Radar Signal Processing with MATLAB Algorithms.* New York, NY: Wiley & Sons, Inc., 1999.

2. Cumming, Ian G and Wong, Frank H. *Digital Processing of Synthetic Aperture Radar Data: Algorithms and Implementations.* Norwood, MA: Artech House, Inc., 2005.

3. Military tanks, vehicles, and artillery. *MilitaryFactory.com*. [Online]. 2003. [Cited: March 2, 2010] http://www.militaryfactory.com/armor/

4. Glossary of remote sensing terms. *Natural Resources Canada.* [Online] Canada Centre for Remote Sensing, November 21, 2005. [Cited: April 5, 2009.] http://www.ccrs.nrcan.gc.ca/glossary/index_e.php?id=2284.

5. Schlutz, Matthew. *Synthetic Aperture Radar Imaging Simulated in Matlab.* San Luis Obispo, CA : California Polytechnic State University San Luis Obispo California, 2009. Master's Thesis.

6. Mason, Paul Ryan. *MATLAB Simulation of Two-Dimensional SAR Imaging By Range Doppler Algorithm.* San Luis Obispo, CA : California Polytechnic State University San Luis Obispo California, 2007. Master's Thesis.

7. Perspective and Orthographic Projection. *Blender*. [Online]. [Cited: November 20, 2009.] http://wiki.blender.org/index.php/Doc:Manual/3D_interaction/Navigating/3D_View.

8. Oleary, Ellen. SEASAT 1978. *Imaging Radar @ Southport.* [Online] Jet Propulsion Laboratories, February 10, 1998. [Cited: April 28, 2009.] http://southport.jpl.nasa.gov/scienceapps/seasat.html.

9. Zaharris, Brian. *Two-Dimensional Synthetic Aperture Radar Imaging and Moving Target Tracking Using the Range Doppler Algorithm Simulated in MATLAB.* San Luis Obispo, CA : California Polytechnic State University San Luis Obispo California, 2006. Master's Thesis.

10. W. Carrara, R. Goodman, and R. Majewski. *Spot light Synthetic Aperture Radar Signal Processing Algorithms.* Artech House. Massachusetts, 1995.

11. Butler, M.J.A., et al. The application of remote sensing technology to marine fisheries: an introductory manual. *Food and Agriculture Organization Coporate Document Repository*. [Online] 1988. [Cited: April 5, 2009.] http://www.fao.org/docrep/003/t0355e/T0355E05.HTM. ISBN: 9251026947.

12. L. Kendrick. *Synthetic Aperture Radar Simulation for Point Target Using MATLAB.* California Polytechnic State University, San Luis Obispo, 2006.

Appendix A: 3-D SAR Simulation Parameters

Symbol	Parameter Name	Value
PRF	Pulse Repetition Frequency	300 Hz
dur	Duration	3 seconds
V_p	Platform Velocity	200 m/s
f_o	Carrier Frequency	4.5 GHz
La	Antenna Actual Length	2 m
X_c	Minimum Range Distance	20 km
X0	Half Target Area Width	200 m
Tp	Chirp Pulse Duration	2.5 μs
B0	Signal Bandwidth	100 MHz
σ_{noise}	AWGN Standard Deviation	0.2 (normalized/unitless)
a_{bins}	Azimuth Bins	900
r_{bins}	Range Bins	1034
	Pixel Separation	0.333 m
	Range Scale	10 m per 1blender unit
	Pixels per pulse	2

Appendix B – Antenna pattern code

File: AntennaPatternGenAppDelegate.m

```
//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////
//  AntennaPatternGenAppDelegate.m
//  AntennaPatternGen
//
//

#import "AntennaPatternGenAppDelegate.h"

@implementation AntennaPatternGenAppDelegate

@synthesize window;

- (void)applicationDidFinishLaunching:(NSNotification *)aNotification {
      // Insert code here to initialize your application
}

@end
//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////
```

File: PatternView.m

```
//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////
//
//  PatternView.m
//  AntennaPatternGen
//
//

#import "PatternView.h"

#import "PatternGenAndSave.h"


@implementation PatternView

- (void)awakeFromNib
{
      imgexposure = 1.f;
      imgwidth = 6000;
      imgheight = 5000;
      imgSeparation = 0.333333f;
      antLength = 2.f;
      frequency = 4.5e9f;
      waveLength = 3e8/frequency;
```

```
      rangeScale = 10.0;
      rangeCenter = 20000.0f;
}

- (void) SettingChange: (id)Sender {
      imgheight = [HeightSetting intValue];
      imgwidth = [WidthSetting intValue];
      imgSeparation = [PixelSepSetting floatValue];
      antLength = [AntennaLengthSetting floatValue];
      waveLength = 3e8/[FrequencySetting floatValue];
      rangeScale = [RangeScaleSetting floatValue];
      rangeCenter = [RangeCenterSetting floatValue];

      if(!data)
            data = (float *)malloc(imgwidth*imgheight*sizeof(float));
      if(!rangeMod)
            rangeMod = (float *)malloc(imgwidth*imgheight*sizeof(float));

      GenPattern(data, rangeMod, imgwidth, imgheight, imgSeparation, antLength,
waveLength,
                        rangeScale, rangeCenter);

      [self setNeedsDisplay:YES];
}

- (void) SizeChange: (id)Sender {
      imgheight = [HeightSetting intValue];
      imgwidth = [WidthSetting intValue];

      if(data) free(data);
      data = (float *)malloc(imgwidth*imgheight*sizeof(float));
      if(rangeMod) free(rangeMod);
      rangeMod = (float *)malloc(imgwidth*imgheight*sizeof(float));

      GenPattern(data, rangeMod, imgwidth, imgheight, imgSeparation,
antLength,waveLength,
                        rangeScale, rangeCenter);

      [self setNeedsDisplay:YES];
}

- (void) ExposureChange: (id)Sender {
      imgexposure = pow(2,[ExposureSetting floatValue]);

      [self setNeedsDisplay:YES];
}



- (void) drawRect:(NSRect)dirtyRect {
      NSRect bds = dirtyRect;

      int myRowBytes = 4 * bds.size.width;
```

```
      if(myDataPtr) free(myDataPtr);
      if(myImageCache) [myImageCache release];

      myDataPtr = (unsigned char *)malloc(myRowBytes * bds.size.height);
      myImageCache = [[NSBitmapImageRep alloc]
initWithBitmapDataPlanes:&myDataPtr

         pixelsWide:bds.size.width

         pixelsHigh:bds.size.height

      bitsPerSample:8

samplesPerPixel:4

             hasAlpha:YES

             isPlanar:NO

colorSpaceName:NSCalibratedRGBColorSpace

       bitmapFormat:0

        bytesPerRow:4*bds.size.width

       bitsPerPixel:32];

      if(data) {
            int i, j;
            float x, y, cap;

            for (j=0; j<bds.size.height; j++) {

                  y = (float)j/bds.size.height;

                  for (i=0; i<bds.size.width; i++) {

                        x = (float)i/bds.size.width;

                        int drawIndex = j*bds.size.width + i;
                        int imgx = x*imgwidth;
                        int imgy = y*imgheight;
                        int imgIndex = imgy*imgwidth + imgx;

                        cap = data[imgIndex]*imgexposure;
                        if(cap > 1) cap = 1;
                        myDataPtr[drawIndex*4] = cap*255;
                        myDataPtr[drawIndex*4+1] = cap*255;
//                      cap = rangeMod[imgIndex]*imgexposure;
//                      if(cap > 1) cap = 1;
                        myDataPtr[drawIndex*4+2] = cap*255;
                        myDataPtr[drawIndex*4+3] = 255;
                  }
            }
```

```
      }

      [myImageCache draw];

}

- (void) SaveImage: (id)Sender {
      NSLog(@"doSaveAs");
      NSSavePanel *tvarNSSavePanelObj = [NSSavePanel savePanel];
      int tvarInt = [tvarNSSavePanelObj runModal];
      if(tvarInt == NSOKButton){
            NSLog(@"doSaveAs we have an OK button");
      } else if(tvarInt == NSCancelButton) {
            NSLog(@"doSaveAs we have a Cancel button");
            return;
      } else {
            NSLog(@"doSaveAs tvarInt not equal 1 or zero = %3d",tvarInt);
            return;
      } // end if
      NSString * tvarDirectory = [tvarNSSavePanelObj directory];
      NSLog(@"doSaveAs directory = %@",tvarDirectory);
      NSString * tvarFilename = [tvarNSSavePanelObj filename];
      NSLog(@"doSaveAs filename = %@",tvarFilename);

      saveEXR([tvarFilename UTF8String], imgwidth, imgheight, data, rangeMod,
                  imgSeparation, frequency, antLength, rangeScale,
rangeCenter);

}


@end
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
```

File: PatternGenAndSave.cpp

```
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
/*
 *  PatternGenAndSave.cpp
 *  AntennaPatternGen
 *
 *
 */

#include "PatternGenAndSave.h"
#include <cmath>

using namespace std;

#define PI 3.14159265358979323846
```

```
#define PLANE_DIST 20000.0
#define PLANE_PIXEL     0.33333333

double sinc(double th){
      if(th == 0)
            return 1.0;
      return sin(th)/th;
}

// This pattern generation function implements a sinc squared function
// Wa = sinc(0.866 * theta / BW)^2
// BW = 0.866lamda/antennaLength
// thus the final sinc(theta * antLenth/waveLength)^2.
// added to this pattern generation is now also a range adjustment.
// Blender output a distance from the plane of the camera to the object.
// To get the range from the point of the camera, the range needs to be
modified
// by the inverse cosine of the angle.
void GenPattern(float *data, float *rangeMod, int width, int height, float
pixelSpacing,
                        float antLenth, float waveLength, float rangeScale,
float rangeCenter){
      int i, j;
      int x, y;

      double radialAngle;

      double xDisplacement, yDisplacement;
      double oppSide;

      // calculate this outside the loop to save loop execution time.
      double syncMod    = antLenth/waveLength;

      for (j=0; j<height; j++) {
            y = j - height/2;
            yDisplacement = abs(y) * pixelSpacing;
            for (i = 0; i < width; i++) {
                  x = i - width/2;
                  xDisplacement = abs(x) * pixelSpacing;


                  oppSide = sqrt(pow(xDisplacement, 2.0) + pow(yDisplacement,
2.0));
                  radialAngle = atan2(oppSide, (double)rangeCenter);
                  double sincRes = sinc(radialAngle * syncMod);
                  // square the sinc result and store in the array.
                  data[i + (j * width)] = pow(sincRes, 2.0);

                  // calculate the range displacement
                  rangeMod[i + (j*width)] = rangeScale/cos(radialAngle);
            }
      }
```

```
}

void saveEXR(const char* filename, int width, int height, const float* data,
const float* rangeMod,
                  float pixelSpacing, float frequency, float antLen, float
rangeScale, float rangeCenter){
      Header header (width, height);
      header.insert("pixel spacing", FloatAttribute (pixelSpacing));
      header.insert("frequency", FloatAttribute (frequency));
      header.insert("antenna length", FloatAttribute (antLen));
      header.insert("range scale", FloatAttribute (rangeScale));
      header.insert("range center", FloatAttribute (rangeCenter));
      header.channels().insert ("G", Channel (FLOAT));
      header.channels().insert ("Z", Channel (FLOAT));

      OutputFile file (filename, header);

      FrameBuffer frameBuffer;

      frameBuffer.insert ("G", Slice(FLOAT, (char *)data, sizeof(*data),
sizeof(*data)*width));
      frameBuffer.insert ("Z", Slice(FLOAT, (char *)rangeMod,
sizeof(*rangeMod),
                                                sizeof(*rangeMod)*width));

      file.setFrameBuffer(frameBuffer);
      file.writePixels(height);
}
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
```

File: AntennaPatternGenAppDelegate.h

```
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
//
//  AntennaPatternGenAppDelegate.h
//  AntennaPatternGen
//
//

#import <Cocoa/Cocoa.h>

@interface AntennaPatternGenAppDelegate : NSObject <NSApplicationDelegate> {
    NSWindow *window;
}

@property (assign) IBOutlet NSWindow *window;
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
```

File: PatternGenAndSave.h

```
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
/*
 *  PatternGenAndSave.h
 *  AntennaPatternGen
 *
 *
 */

#ifndef PATTERNGENANDSAVE_H
#define PATTERNGENANDSAVE_H

#ifdef __cplusplus

#include <ImfRgbaFile.h>
#include <ImfChannelList.h>
#include <ImfHeader.h>
#include <ImfOutputFile.h>
#include <ImfInputFile.h>
#include <ImfStandardAttributes.h>
#include <ImfArray.h>

using namespace Imf;
using namespace Imath;

#endif

#ifdef __cplusplus
extern "C" {
#endif

      void GenPattern(float *data, float *rangeMod, int width, int height,
float pixelSpacing,
                              float antLength, float waveLength, float
rangeScale, float rangeCenter);
      void saveEXR(const char* filename, int width, int height, const float*
data,
                           const float* rangeMod, float pixelSpacing, float
frequency, float antLen,
                           float rangeScale, float rangeCenter);

#ifdef __cplusplus
}
#endif

#endif //PATTERNGENANDSAVE_H
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
```

File: PatternView.h

```
//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////
//
//  PatternView.h
//  AntennaPatternGen
//
//

#import <Cocoa/Cocoa.h>


@interface PatternView : NSView {
```

File: PatternView.h

```
//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////
//
//  PatternView.h
//  AntennaPatternGen
//
//

#import <Cocoa/Cocoa.h>


@interface PatternView : NSView {
int imgwidth, imgheight;
      float imgSeparation;
      float antLength;
      float waveLength;
      float imgexposure;
      float frequency;
      float rangeScale;
      float rangeCenter;
```

Appendix C – Echo generator code

File: main.m

```
//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////
//
//  main.m
//  EchoGenerator
//
//

#import <Cocoa/Cocoa.h>

int main(int argc, char *argv[])
{
    return NSApplicationMain(argc,  (const char **) argv);
}
//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////
```

File: EchoGeneratorAppDelegate.m

```
//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////
//  EchoGeneratorAppDelegate.m
//  EchoGenerator
//
//

#import "EchoGeneratorAppDelegate.h"
#include "loadimage.h"
#include "logging.h"


@implementation EchoGeneratorAppDelegate

@synthesize window;

- (void)applicationDidFinishLaunching:(NSNotification *)aNotification {
      // Insert code here to initialize your application
      [self EchoSetupChange: self];
      imageToEchoSetup.height = 2880;
      imageToEchoSetup.width  = 2880;
      imageToEchoSetup.carrierFreq = 4.5e9f;
      imageToEchoSetup.rangeScale = 10.0;
      imageToEchoSetup.rangeCenter = 20000.0f;
      imageToEchoSetup.pixelSeparation = 0.3333f;

      antennaPattern = NULL;
      imageData = NULL;
```

```
      rangeData = NULL;
      quadDemodSignal = NULL;
      echo = NULL;

      imgexposure = 1.0;

}

- (IBAction) EchoSetupChange: (id)Sender{
      float rangeDelta = [rangeWidth floatValue];
      float pulseFreqVar = [pulseRepetitionFreq floatValue];
      float pixPerPulse = [PixelsPerPulse floatValue];
      int echoBinCalc;

      imageToEchoSetup.simDuration = [SimDuration floatValue];

      imageToEchoSetup.rangeWidth = rangeDelta;
      imageToEchoSetup.pulseFreq = pulseFreqVar;
      imageToEchoSetup.pixPerPulse = pixPerPulse;
      imageToEchoSetup.basebandBW = [basebandBW floatValue];
      imageToEchoSetup.pulseDuration = [pulseDur floatValue];
      echoBinCalc = ceil(((2.0 * (rangeDelta + 1) / SPEED_OF_LIGHT) +
                                    imageToEchoSetup.pulseDuration) * (2.0 *
imageToEchoSetup.basebandBW));
      // Not sure why, but the effect of some of the matlab code is to round
this figure to an
      // even value. To match this behaviour, the following line is added.
      // Matlab code is now understood. FFT functions cannot handle odd number
of data items.
      echoBinCalc += echoBinCalc & 0x1;
      imageToEchoSetup.numEchoBins = echoBinCalc;
      imageToEchoSetup.numQuadDemodSamples =
ceil(imageToEchoSetup.pulseDuration * 2.0 *

      imageToEchoSetup.basebandBW) + 1;

      [echoBins setIntValue:echoBinCalc];
      [demodSamples  setIntValue:imageToEchoSetup.numQuadDemodSamples];
      [Velocity setFloatValue:(pixPerPulse * pulseFreqVar *
imageToEchoSetup.pixelSeparation)];

}

- (IBAction) LoadAntennaPattern: (id)Sender{

//    LogInfo(@"Entered");
      NSOpenPanel *panel = [NSOpenPanel openPanel];
      [panel setAllowedFileTypes:[NSArray arrayWithObject:@"exr"]];
      [panel setAllowsOtherFileTypes:YES];
      [panel setAllowsMultipleSelection:NO];
      [panel setTitle:[NSString stringWithUTF8String:"Select Antenna Pattern
File"]];
      NSInteger panReturn = [panel runModal];
```

```
      if (panReturn == NSOKButton){
//          LogInfo(@"OK Button");
      } else if (panReturn == NSCancelButton) {
//          LogInfo(@"Cancel Button");
            return;
      } else {
//          LogInfo(@"Unrecognized return = %3d", panReturn);
            return;
      }

      NSURL *openFile = [panel URL];
//    LogInfo(@"filename = %@", openFile);

      int imgHeight, imgWidth;
      int err = GetImageSize([[openFile path] UTF8String], &imgWidth,
&imgHeight);

      if (err < 0) {
//          LogError(@"Error return from GetImageSize = %d", err);
            return;
      }

      if (antennaPattern != NULL){
            delete[] antennaPattern;
            antennaPattern = NULL;
      }

      antennaPattern = new float[imgWidth * imgHeight];

      if (rangeMod != NULL){
            delete[] rangeMod;
            rangeMod = NULL;
      }

      rangeMod = new float[imgWidth * imgHeight];

      [height setIntegerValue: imgHeight];
      [width setIntegerValue: imgWidth];
      imageToEchoSetup.height = imgHeight;
      imageToEchoSetup.width = imgWidth;

      float carrier;
      float pixelSeparation;
      float antLength;
      float rScaleVar;
      float rCenterVar;

      err = LoadAntennaPattern([[openFile path] UTF8String], imgWidth,
imgHeight,
                                          antennaPattern, rangeMod, &carrier,
&pixelSeparation, &antLength,
                                          &rScaleVar, &rCenterVar);

      if (err < 0) {
```

```
//          LogError(@"Error return from LoadAntennaPattern = %d", err);
            return;
      }

      imageToEchoSetup.carrierFreq = carrier;
      imageToEchoSetup.antLength = antLength;
      imageToEchoSetup.pixelSeparation = pixelSeparation;
      imageToEchoSetup.rangeScale = rScaleVar;
      imageToEchoSetup.rangeCenter = rCenterVar;

      [carrierFreq setFloatValue:carrier];
      [rangeScale setFloatValue:rScaleVar];
      [rangeCenter setFloatValue:rCenterVar];
      [pixelSpacing setFloatValue:pixelSeparation];
      [Velocity setFloatValue:(pixelSeparation * imageToEchoSetup.pixPerPulse *
                                          imageToEchoSetup.pulseFreq)];
}

NSInteger MyCompareUrl(NSURL *num1, NSURL *num2, void *context) {
      NSString *num1Path = [num1 path];
      NSString *num2Path = [num2 path];

      return [num1Path caseInsensitiveCompare: num2Path];
}

- (NSInteger) DisplayEcho:(std::complex<float> *)echoData height:(int)
imgheight width:(int) imgwidth{
      NSBitmapImageRep * image = [[NSBitmapImageRep alloc]
initWithBitmapDataPlanes:NULL

                           pixelsWide:imgwidth

                           pixelsHigh:imgheight

                        bitsPerSample:8

                      samplesPerPixel:4

                                hasAlpha:YES

                                isPlanar:NO

                       colorSpaceName:NSCalibratedRGBColorSpace

                           bitmapFormat:0

                            bytesPerRow:4 * imgwidth

                           bitsPerPixel:32];
      int i, j;
      std::complex<float> pixel;
      unsigned char *imgdata = [image bitmapData];
      float real, imaginary;
      for (j=0; j<imgheight; j++) {
            for (i=0; i<imgwidth; i++) {
```

```
                  int drawIndex = j * imgwidth + i;

                  pixel = echoData[drawIndex];
                  real = pixel.real();
                  imaginary = pixel.imag();

                  real *= imgexposure;
                  imaginary *= imgexposure;
                  if(real < 0.0) real = 0.0;
                  if(imaginary < 0.0) imaginary = 0.0;
                  if(real > 1.0) real = 1.0;
                  if(imaginary > 1.0) imaginary = 1.0;

                  imgdata[drawIndex*4] = real*255;
                  imgdata[drawIndex*4+1] = imaginary*255;
//                imgdata[drawIndex*4+1] = 0;
                  imgdata[drawIndex*4+2] = 0;
                  imgdata[drawIndex*4+3] = 255;
            }
      }

      [echoImage SetDisplayImage:image];

      return 0;
}

- (void) ExposureChange: (id)Sender {
      imgexposure = pow(2,[ExposureSetting floatValue]);

      [self DisplayEcho:echo height: numPulses width:
imageToEchoSetup.numEchoBins];
}

- (IBAction) LoadRangeSamples: (id)Sender{

      if (antennaPattern == NULL) {
//          LogError(@"Antenna Pattern Data not loaded!");
            return;
      }

      int imgHeight = imageToEchoSetup.height;
      int imgWidth = imageToEchoSetup.width;
      numPulses = imageToEchoSetup.simDuration;

      imgWidth = imgWidth + (imageToEchoSetup.pixPerPulse *
imageToEchoSetup.simDuration);
      imageToEchoSetup.imgWidth = imgWidth;

      if (imageData != NULL) {
            delete[] imageData;
            imageData = NULL;
      }
      if (rangeData != NULL) {
            delete[] rangeData;
            rangeData = NULL;
```

```
      }
      imageData = new float[imgWidth * imgHeight];
      rangeData = new float[imgWidth * imgHeight];

      if (quadDemodSignal != NULL) {
            delete[] quadDemodSignal;
            quadDemodSignal = NULL;
      }
      quadDemodSignal = new
std::complex<float>[imageToEchoSetup.numQuadDemodSamples];

      int err = QuadDemodXmitSignalCalculate(&imageToEchoSetup,
quadDemodSignal);

      NSOpenPanel *panel = [NSOpenPanel openPanel];
      [panel setAllowedFileTypes:[NSArray arrayWithObject:@"exr"]];
      [panel setAllowsOtherFileTypes:YES];
      [panel setAllowsMultipleSelection:NO];
      [panel setTitle:[NSString stringWithUTF8String:"Select Platform Track
Images"]];
      NSInteger panelReturn = [panel runModal];

      if (panelReturn == NSOKButton) {
//          LogInfo(@"OK Button");
      } else if (panelReturn == NSCancelButton) {
//          LogInfo(@"Cancel Button");
            return;
      } else {
//          LogInfo(@"Unrecognized Return = %3d", panelReturn);
            return;
      }

//    NSArray *alphabeticalFileList = [panel URLs];
//    NSArray *alphabeticalFileList = [[panel URLs]
sortedArrayUsingFunction:MyCompareUrl context:NULL];

      if (echo != NULL) {
            delete[] echo;
            echo = NULL;
      }
      int numBins = imageToEchoSetup.numEchoBins;
      echo = new std::complex<float>[numPulses * numBins];

      NSString *filePath = [[panel URL] path];
//    NSString *filePath = [[alphabeticalFileList objectAtIndex:1] path];
      //          LogInfo(@"Processing File: %@", filePath);
      err = LoadImageData([filePath UTF8String],
                                    imgWidth, imgHeight, imageData, rangeData);
      if (err < 0) {
            //                LogError(@"LoadImageData returned an error = %d",
err);
            return;
      }
      for (int i = 0; i < numPulses; i++) {
```

```
            err = ImageToEcho(&imageToEchoSetup, antennaPattern, rangeMod,
imageData, rangeData,
                                        quadDemodSignal, &echo[i * numBins], i);
            if (err < 0) {
//                LogError(@"ImageToEcho returned an error = %d", err);
                  return;
            }

      }
//    LogInfo(@"Echo Generation complete! Displaying results.");
//    numFiles++;
      [self DisplayEcho:echo height: numPulses width: numBins];
}

- (IBAction) SaveEchoData: (id)Sender {
      if(echo == NULL)
            return;

      float settings[SETTING_TOTAL_NUMBER];

      settings[SETTING_RANGE_CENTER_OFFSET] = imageToEchoSetup.rangeCenter;
      settings[SETTING_RANGE_WIDTH_OFFSET] = imageToEchoSetup.rangeWidth;
      settings[SETTING_CARRIER_FREQ_OFFSET] = imageToEchoSetup.carrierFreq;
      settings[SETTING_BASEBAND_BW_OFFSET] = imageToEchoSetup.basebandBW;
      settings[SETTING_PULSE_DURATION_OFFSET] = imageToEchoSetup.pulseDuration;
      settings[SETTING_RANGE_SCALE_OFFSET] = imageToEchoSetup.rangeScale;
      settings[SETTING_ANTENNA_LENGTH_OFFSET] = imageToEchoSetup.antLength;
      settings[SETTING_PULSE_REP_FREQ_OFFSET] = imageToEchoSetup.pulseFreq;
      settings[SETTING_PIXEL_SPACING_OFFSET] =
imageToEchoSetup.pixelSeparation;
      settings[SETTING_PIX_PER_PULSE_OFFSET] = imageToEchoSetup.pixPerPulse;

      NSLog(@"doSaveAs");
      NSSavePanel *tvarNSSavePanelObj = [NSSavePanel savePanel];
      int tvarInt = [tvarNSSavePanelObj runModal];
      if(tvarInt == NSOKButton){
            NSLog(@"doSaveAs we have an OK button");
      } else if(tvarInt == NSCancelButton) {
            NSLog(@"doSaveAs we have a Cancel button");
            return;
      } else {
            NSLog(@"doSaveAs tvarInt not equal 1 or zero = %3d",tvarInt);
            return;
      } // end if
      NSString * tvarDirectory = [tvarNSSavePanelObj directory];
      NSLog(@"doSaveAs directory = %@",tvarDirectory);
      NSString * tvarFilename = [tvarNSSavePanelObj filename];
      NSLog(@"doSaveAs filename = %@",tvarFilename);

      SaveEchoData([tvarFilename UTF8String], imageToEchoSetup.numEchoBins,
numPulses, echo, settings);

}
```

```
@end
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
```

File: EchoViewer.m

```
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
//
//  EchoViewer.m
//  EchoGenerator
//
//

#import "EchoViewer.h"


@implementation EchoViewer

#define IMG_HEIGHT 900
#define IMG_WIDTH 1023

- (id)initWithFrame:(NSRect)frame {
    self = [super initWithFrame:frame];
    if (self) {
        // Initialization code here.
            displayImage = [[NSBitmapImageRep alloc]
initWithBitmapDataPlanes:NULL

               pixelsWide:IMG_WIDTH

               pixelsHigh:IMG_HEIGHT

           bitsPerSample:8

        samplesPerPixel:4

                   hasAlpha:YES

                   isPlanar:NO

         colorSpaceName:NSCalibratedRGBColorSpace

              bytesPerRow:4*IMG_WIDTH

             bitsPerPixel:32];
           [self setNeedsDisplay:YES];
    }
    return self;
}
```

```
- (void)drawRect:(NSRect)dirtyRect {
    // Drawing code here.
      [displayImage draw];
}

- (void)SetDisplayImage:(NSBitmapImageRep *)image {
      if(displayImage){
            [displayImage dealloc];
      }
      displayImage = image;

      [self setNeedsDisplay:YES];
}

@end
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
```

File: imagetoecho.cpp

```
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
/*
 *  imagetoecho.cpp
 *  EchoGenerator
 *.
 *
 */

#include "imagetoecho.h"
#include "logging.h"

#include <complex>


#define PI 3.14159265358979323846264338327795

/* function to take details about the transmitted SAR pulse, and calculate the
output of
 * the quadrature demodulation circuit based on an ideal return.
 * The setup structure contains data about the simulation parameters.
 * The signal array pointer is a pointer to an array of the size of the
numQuadDemodSamples
 * element of the setup structure.
 * This value must be greater than or equal to the value defined by the
 * length of the transmitted pulse multiplied by the sampling frequency. If the
array length
 * is not greater than this value, this function will return -1;
 * The arrayLen value states the length of the array pointed to by signal.
```

```
 */
int QuadDemodXmitSignalCalculate(IMAGE_TO_ECHO_SETUP *setup,
std::complex<float> *signal){
      float pulseDur = setup->pulseDuration;
      // The signal chirp rate is determined by the desired bandwidth, and how
long the pulse is
      // transmitted.
      float chirpRate = setup->basebandBW / pulseDur;
      // To properly sample the data from the quadrature demodulation, there
needs to be as many
      // samples as the frequency range. The baseband bandwidth is a +/- from
center, so
      // multiply by 2.
      float sampleTime = 1.0 / (2.0 * setup->basebandBW);
      // The number of samples on the ideal reflection is determined by the
number of
      // samples that occur during the transmitted pulse width.
      int numSamples = (int)(pulseDur / sampleTime);

      // i * PI * chirpRate is a portion of the exponential calculation for the
quad demod
      // function. By precalculating it outside the loop, we reduce execution
time.
      std::complex<float> piKr = std::complex<float>(0.0, PI * chirpRate);

      float timeOffset;

      int arrayLen = setup->numQuadDemodSamples;

      for (int i=0; i < arrayLen; i++) {
            if (i<numSamples) {
                  timeOffset = i * sampleTime;
                  // calculate the value
                  signal[i] = exp(piKr * ((float)pow(timeOffset, 2.0) -
(timeOffset * pulseDur)));
            } else {
                  signal[i] = std::complex<float>(0.0, 0.0);
            }

      }
      // check to inform the calling function that the array was not large
enough to hold
      // the entire signal demodulation signature.
      if (arrayLen < numSamples){
//          LogDebug(@"arrayLen (%d) is smaller than the number of samples
(%d).",
//                         arrayLen, numSamples);
            return -1;
      }

      return 0;
}

/* function to take an antenna radiance pattern, an image with reflectance
```

```
information
 * and a set of range information, and a demodulation signature, and create an
echo.
 * The antenna radiance pattern, the image, and the range data are two
dimensional arrays
 * of the same size, defined by the width and height fields of the setup
structure.
 * The quadDemodSignal array is a one dimensional array containing the
demodulation
 * signature of the transmitted pulse. This array size is contained in the
numQuadDemodSamples
 * field of the setup data. The array is multiplied by target
location/reflectance as part
 * of creating the echo signal.
 * The echo array is a one dimensional array of the size defined by the
numEchoBins field
 * of the setup structure.
 */
int ImageToEcho(IMAGE_TO_ECHO_SETUP *setup, float *antennaPattern, float
*rangeMod, float *image,
                        float *range, std::complex<float> *quadDemodSignal,
std::complex<float> *echo,
                        int pulseNum){
      int height = setup->height;
      int width = setup->width;
      int imgWidth = setup->imgWidth;

      int reflectLen = setup->numEchoBins + setup->numQuadDemodSamples;

      // The reflection timeline must contain reflections that start before the
      // receiver if the reflection itself would continue into the area where
the receiver
      // is active. Any signal with this characteristic acts as a noise input
into the
      // start of matched filtering.
      std::complex<float> reflection[reflectLen];
      std::complex<float> zero(0.0, 0.0);

      // the rangeTime variable here is the distance light travels for each
sample in the
      // range direction. The calculation calculates the inverse, so we can
multiply in the loop.
      // Multiplication is generally less computationally intensive than
division.
      // The sampling time is multiplied by 2 in order to take into account
that the light goes
      // out and back, thus needing to cover 2X the range that we are
measuring.
      float rangeTime = 2.0 * 2.0 * setup->basebandBW / SPEED_OF_LIGHT;

      std::complex<float> reflectMagnitude;
      int index;
```

```
      // get the ranges where the receiver gets data. Read comment above
reflection for an
      // explanation of why the min range is modified.
      float minRange = setup->rangeCenter - (setup->rangeWidth/2);
      float maxRange = setup->rangeCenter + (setup->rangeWidth/2);
      minRange = (minRange - (setup->numQuadDemodSamples / rangeTime));

      std::complex<float> fourPiF0DivByC = std::complex<float>(0.0,
                                                               -4.0 * PI *
(float)setup->carrierFreq / SPEED_OF_LIGHT);

      int pulseOffset = pulseNum * setup->pixPerPulse;
      for(int j=0; j<height; j++){
            int vertOffset = (j * width);
            int vertImgOffset = (j * imgWidth) + pulseOffset;
            for (int i=0; i<width; i++) {
                  // check if the target is within the range that the receiver
is receiving data
                  float curRange = range[vertImgOffset] * rangeMod[vertOffset];
                  if ((curRange > minRange) && (curRange < maxRange)) {
                        if((image[vertImgOffset] > 1.0f) ||
(image[vertImgOffset] < 0.0f)){
//                              LogError(@"image pixel %d has a value (%f)
outside limits", i, image[i]);
                              return -2;
                        }
                        // calculate the magnitude of the reflection
                        reflectMagnitude = (float)(antennaPattern[vertOffset] *
image[vertImgOffset]) *

      exp(fourPiF0DivByC * (float)(curRange));
                        // determine when the reflection happens on the time
line
                        index = round((curRange - minRange) * rangeTime);
                        if (index > reflectLen) {
//                              LogError(@"index (%d) ended up longer than
reflectLen (%d)!", index, reflectLen);
                              return -1;
                        } else {
                              // add that reflection to the bin.
                              reflection[index] += reflectMagnitude;
                        }
                  }
                  vertOffset++;
                  vertImgOffset++;
            }
      }

      int sigLen = setup->numQuadDemodSamples;

      // multiply the reflection data by the demodulation signature
      // check each location
      for (int location = 0; location < reflectLen; location++){
            if(reflection[location] != zero){
```

```
                  // and multiply by each signature location
                  for (int sigLoc = 0; sigLoc < sigLen; sigLoc++){
                        int echoLoc = sigLoc + location;
                        // calculate if the signature+location places the
multiplication value within
                        // the receiver window. Part of the location
information is previous to the
                        // window as mentioned in the reflection comment, and
part extends beyond the window.
                        if ((echoLoc < reflectLen) && (echoLoc >= sigLen)){
                              // add the echo signature into the final echo
value. multiple signals will overlap
                              // into the echo locations, so the value must be
added to the signal already in the bin.
                              echo[echoLoc - sigLen] += reflection[location] *
quadDemodSignal[sigLoc];
                        }
                  }
            }
      }
      return 0;
}
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
```

File: loadimage.cpp

```
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
/*
 *  loadimage.cpp
 *  EchoGenerator
 *
 *
 */
#include "loadimage.h"
//#include "logging.h"

#include <ImfRgbaFile.h>
#include <ImfChannelList.h>
#include <ImfHeader.h>
#include <ImfOutputFile.h>
#include <ImfInputFile.h>
#include <ImfStandardAttributes.h>
#include <ImfArray.h>

using namespace Imf;
using namespace Imath;

/* function to get the width and height of an OpenEXR image.
```

```
 */
int GetImageSize(const char *filename, int *width, int *height){
      InputFile file(filename);
      Box2i dw = file.header().dataWindow();
      *width  = dw.max.x - dw.min.x + 1;
      *height = dw.max.y - dw.min.y + 1;

      return 0;
}


/* function loads the antenna pattern from a file into an array.
 * returns an error if width and height are not equal to the image header data.
 */
int LoadAntennaPattern(const char *filename, int width, int height, float
*data, float *rangeMod,
                                 float *carrier, float *pixelSep, float
*antLength, float *rangeScale,
                                 float *rangeCenter){
      // open the file, and get the internal window size structure.
      InputFile file(filename);
      Box2i dw = file.header().dataWindow();

      // check the size of the file versus the arguments passed into the
function
      // return an error if they don't match.
      if ((dw.max.x - dw.min.x + 1 != width) || (dw.max.y - dw.min.y + 1 !=
height)) {
            //          LogError(@"image height and width don't match given
dimensions.");
            return -1;
      }

      const FloatAttribute *frequency = file.header().findTypedAttribute

      <FloatAttribute> ("frequency");
      if (frequency == NULL){
//          LogError(@"frequency attribute not found in the file");
            return -2;
      }

      *carrier = frequency->value();

      const FloatAttribute *pixelSpace = file.header().findTypedAttribute

      <FloatAttribute> ("pixel spacing");
      if (pixelSpace == NULL){
            //          LogError(@"Pixel Spacing attribute not found in the
file");
            return -3;
      }

      *pixelSep = pixelSpace->value();
```

```
      const FloatAttribute *antennaLen = file.header().findTypedAttribute

      <FloatAttribute> ("antenna length");
      if (antennaLen == NULL){
            //          LogError(@"antenna length attribute not found in the
file");
            return -4;
      }

      *antLength = antennaLen->value();

      const FloatAttribute *rangeScaleAttr = file.header().findTypedAttribute

      <FloatAttribute> ("range scale");
      if (rangeScaleAttr == NULL){
            //          LogError(@"range scale attribute not found in the
file");
            return -5;
      }

      *rangeScale = rangeScaleAttr->value();

      const FloatAttribute *rangeCenterAttr = file.header().findTypedAttribute
      <FloatAttribute> ("range center");
      if (rangeCenterAttr == NULL){
            //          LogError(@"range center attribute not found in the
file");
            return -6;
      }

      *rangeCenter = rangeCenterAttr->value();

      // create the framebuffer class for IMF
      FrameBuffer frameBuffer;
      frameBuffer.insert ("G",                                    // name
                                    Slice (FLOAT,                           //
type
                                             (char *) data,
                                             sizeof (*data) * 1,    // xStride
                                             sizeof (*data) * (width),//
yStride
                                             1, 1,                          //
x/y sampling
                                             0.0));                         //
fillValue

      frameBuffer.insert ("Z",                                    // name
                                    Slice (FLOAT,                           //
type
                                             (char *) rangeMod,
                                             sizeof (*rangeMod) * 1,    //
xStride
```

```
                                                sizeof (*rangeMod) * (width),//
yStride
                                                1, 1,                          //
x/y sampling
                                                0.0));                         //
fillValue

      file.setFrameBuffer (frameBuffer);
      // read the data from the file
      file.readPixels (dw.min.y, dw.max.y);

      return 0;
}


/* function loads image data from an OpenEXR file.
 * loads RGB data, and sets to zero if R, G, and B are not the same.
 * loads the Z data into the range buffer.
 * function returns an error if width and height are not equal to the image
header width
 * and height.
 */
int LoadImageData(const char *filename, int width, int height, float *data,
float *range){
      // open the file, and get the internal window size structure.
      InputFile file(filename);
      Box2i dw = file.header().dataWindow();

      // check the size of the file versus the arguments passed into the
function
      // return an error if they don't match.
      if ((dw.max.x - dw.min.x + 1 != width) || (dw.max.y - dw.min.y + 1 !=
height)) {
//          LogError(@"image height and width don't match given dimensions.");
            return -1;
      }

      // create the framebuffer class for IMF
      FrameBuffer frameBuffer;
      frameBuffer.insert ("G",                                  // name
                                    Slice (FLOAT,                          //
type
                                                (char *) data,
                                                sizeof (data[0]) * 1,    //
xStride
                                                sizeof (data[0]) * (width),//
yStride
                                                1, 1,                          //
x/y sampling
                                                0.0));                         //
fillValue

      frameBuffer.insert ("Z",                                  // name
```

```
                                    Slice (FLOAT,                          //
type
                                           (char *) range,
                                           sizeof (range[0]) * 1,    //
xStride
                                           sizeof (range[0]) * (width),//
yStride
                                           1, 1,                          //
x/y sampling
                                           0.0));                         //
fillValue

      file.setFrameBuffer (frameBuffer);
      // read the data from the file
      file.readPixels (dw.min.y, dw.max.y);

      return 0;

}

/* function saves echo data to an OpenEXR file.
 * saves data as floating point values, real in channel "R", and imaginary in
channel "G"
 * settings get included in the EXR header as float attributes. The name of the
float
 * attribute is based on the constant names defined in the header file.
 */
int SaveEchoData(const char *filename, int width, int height,
std::complex<float> *data,
                        float *settings){

      Header header (width, height);
      if(settings != NULL){
            header.insert("range center", FloatAttribute
(settings[SETTING_RANGE_CENTER_OFFSET]));
            header.insert("range width", FloatAttribute
(settings[SETTING_RANGE_WIDTH_OFFSET]));
            header.insert("frequency", FloatAttribute
(settings[SETTING_CARRIER_FREQ_OFFSET]));
            header.insert("baseband bw", FloatAttribute
(settings[SETTING_BASEBAND_BW_OFFSET]));
            header.insert("pulse duration", FloatAttribute
(settings[SETTING_PULSE_DURATION_OFFSET]));
            header.insert("range scale", FloatAttribute
(settings[SETTING_RANGE_SCALE_OFFSET]));
            header.insert("antenna length", FloatAttribute
(settings[SETTING_ANTENNA_LENGTH_OFFSET]));
            header.insert("pulse repetition freq",
                              FloatAttribute
(settings[SETTING_PULSE_REP_FREQ_OFFSET]));
            header.insert("pixel spacing", FloatAttribute
(settings[SETTING_PIXEL_SPACING_OFFSET]));
            header.insert("pixels per pulse", FloatAttribute
(settings[SETTING_PIX_PER_PULSE_OFFSET]));
```

```
      }
      header.channels().insert ("R", Channel (FLOAT));
      header.channels().insert ("G", Channel (FLOAT));

      float rpix[width*height];
      float gpix[width*height];

      for(int i=0; i<width*height; i++){
            rpix[i] = data[i].real();
            gpix[i] = data[i].imag();
      }

      OutputFile file (filename, header);

      FrameBuffer frameBuffer;

      frameBuffer.insert ("G", Slice(FLOAT, (char *)gpix, sizeof(*gpix),
sizeof(*gpix)*width));
      frameBuffer.insert ("R", Slice(FLOAT, (char *)rpix, sizeof(*rpix),
sizeof(*rpix)*width));

      file.setFrameBuffer(frameBuffer);
      file.writePixels(height);


      return 0;
}
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
```

File: EchoGeneratorAppDelegate.h

```
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
//
//  EchoGeneratorAppDelegate.h
//  EchoGenerator
//.
//
#import <Cocoa/Cocoa.h>
#import "imagetoecho.h"
#import "EchoViewer.h"

@interface EchoGeneratorAppDelegate : NSObject <NSApplicationDelegate> {
    NSWindow *window;

      IMAGE_TO_ECHO_SETUP imageToEchoSetup;
      float *antennaPattern;
      float *rangeMod;
      float *imageData;
      float *rangeData;
      std::complex<float> *quadDemodSignal;
      std::complex<float> *echo;
      double imgexposure;
```

```
      int numPulses;

      IBOutlet id rangeWidth;
      IBOutlet id pulseRepetitionFreq;
      IBOutlet id carrierFreq;
      IBOutlet id basebandBW;
      IBOutlet id pulseDur;
      IBOutlet id rangeScale;
      IBOutlet id rangeCenter;
      IBOutlet id echoBins;
      IBOutlet id demodSamples;
      IBOutlet id height;
      IBOutlet id width;
      IBOutlet id ExposureSetting;
      IBOutlet id pixelSpacing;
      IBOutlet id PixelsPerPulse;
      IBOutlet id Velocity;
      IBOutlet id SimDuration;
      IBOutlet EchoViewer *echoImage;
}
@property (assign) IBOutlet NSWindow *window;

- (IBAction) EchoSetupChange: (id)Sender;
- (IBAction) LoadAntennaPattern: (id)Sender;
- (IBAction) LoadRangeSamples: (id)Sender;
- (IBAction) SaveEchoData: (id)Sender;
- (IBAction) ExposureChange: (id)Sender;

@end
//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////
```

File: EchoViewer.h

```
//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////
//  EchoViewer.h
//  EchoGenerator
//
//

#import <Cocoa/Cocoa.h>


@interface EchoViewer : NSView {
      NSBitmapImageRep *displayImage;
}

- (void) drawRect:(NSRect)dirtyRect;
- (void) SetDisplayImage: (NSBitmapImageRep *)image;

@end
//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////
```

File: imagetoecho.h

```
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
/*
 *  imagetoecho.h
 *  EchoGenerator
 *
 *
 */

#ifndef IMAGE_TO_ECHO_H
#define IMAGE_TO_ECHO_H

#define SPEED_OF_LIGHT 299792458.0

#include <complex>

typedef struct {
      int height;
      int width;
      int imgWidth;
      int numEchoBins;
      int numQuadDemodSamples;
      int simDuration;
      float rangeWidth;
      float rangeCenter;
      float carrierFreq;
      float basebandBW;
      float pulseDuration;
      float rangeScale;
      float antLength;
      float pulseFreq;
      float pixelSeparation;
      float pixPerPulse;
} IMAGE_TO_ECHO_SETUP;

/* function to take details about the transmitted SAR pulse, and calculate the
output of
 * the quadrature demodulation circuit based on an ideal return.
 * The setup structure contains data about the simulation parameters.
 * The signal array pointer is a pointer to an array of the size defined by the
 * length of the transmitted pulse multiplied by the sampling frequency.
 */
int QuadDemodXmitSignalCalculate(IMAGE_TO_ECHO_SETUP *setup,
std::complex<float> *signal);

/* function to take an antenna radiance pattern, an image with reflectance
information,
 * and a set of range information, and create an echo.
 * The antenna radiance pattern, the image, and the range data are two
dimensional arrays
 * of the same size, defined by the width and height fields of the setup
structure.
 * The echo array is a one dimensional array of the size defined by the
```

```
numEchoBins field
 * of the setup structure.
 * This function does not allocate memory for any of the arguments. They must
have been
 * allocated outside this function and passed in. It will also not free any of
these
 * arguments.
 * For internal use, the function will allocate a buffer the size of the image,
and delete
 * it before returning.
 */
int ImageToEcho(IMAGE_TO_ECHO_SETUP *setup, float *antennaPattern, float
*rangeMod, float *image,
                        float *range, std::complex<float> *quadDemodSignal,
std::complex<float> *echo,
                        int pulseNum);


#endif // IMAGE_TO_ECHO_H
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
```

File: loadimage.h

```
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
/*
 *  loadimage.h
 *  EchoGenerator
 *
 *
 */

#ifndef LOAD_IMAGE_H
#define LOAD_IMAGE_H

#include <complex>

#define SETTING_RANGE_CENTER_OFFSET                 0
#define SETTING_RANGE_WIDTH_OFFSET                  1
#define SETTING_CARRIER_FREQ_OFFSET                 2
#define SETTING_BASEBAND_BW_OFFSET                  3
#define SETTING_PULSE_DURATION_OFFSET               4
#define SETTING_RANGE_SCALE_OFFSET                  5
#define SETTING_ANTENNA_LENGTH_OFFSET               6
#define SETTING_PULSE_REP_FREQ_OFFSET               7
#define SETTING_PIXEL_SPACING_OFFSET                8
#define SETTING_PIX_PER_PULSE_OFFSET                9


#define SETTING_TOTAL_NUMBER                        10
```

```
/* function to get the width and height of an OpenEXR image.
 */
int GetImageSize(const char *filename, int *width, int *height);

/* function loads the antenna pattern from a file into an array.
 * returns an error if width and height are not equal to the image header data.
 */
int LoadAntennaPattern(const char *filename, int width, int height, float
*data, float *rangeMod,
                                   float *carrierFreq, float *pixelSep, float
*antLength, float *rangeScale,
                                   float *rangeCenter);

/* function loads image data from an OpenEXR file.
 * loads RGB data, and sets to zero if R, G, and B are not the same.
 * loads the Z data into the range buffer.
 * function returns an error if width and height are not equal to the image
header width
 * and height.
 */
int LoadImageData(const char *filename, int width, int height, float *data,
float *range);

/* function saves echo data to an OpenEXR file.
 * saves data as floating point values, real in channel "R", and imaginary in
channel "G"
 * settings get included in the EXR header as float attributes. The name of the
float
 * attribute is based on the constant names defined in the header file.
 */
int SaveEchoData(const char *filename, int width, int height,
std::complex<float> *data,
                         float *settings);

#endif //LOAD_IMAGE_H
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
```

File: logging.h

```
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////
/*
 *  logging.h
 */

/*
 * There are three levels of logging: debug, info and error, and each can be
enabled independently
 * via the LOGGING_LEVEL_DEBUG, LOGGING_LEVEL_INFO, and LOGGING_LEVEL_ERROR
switches below, respectively.
 * In addition, ALL logging can be enabled or disabled via the LOGGING_ENABLED
switch below.
 *
 * To perform logging, use any of the following function calls in your code:
 *
 * LogDebug(fmt, ...) â€“ will print if LOGGING_LEVEL_DEBUG is set on.
 * LogInfo(fmt, ...) â€“ will print if LOGGING_LEVEL_INFO is set on.
 * LogError(fmt, ...) â€“ will print if LOGGING_LEVEL_ERROR is set on.
 *
 * Each logging entry can optionally automatically include class, method and
line information by
 * enabling the LOGGING_INCLUDE_CODE_LOCATION switch.
 *
 * Logging functions are implemented here via macros, so disabling logging,
either entirely,
 * or at a specific level, removes the corresponding log invocations from the
compiled code,
 * thus completely eliminating both the memory and CPU overhead that the
logging calls would add.
 */

// Set this switch to enable or disable ALL logging.
#define LOGGING_ENABLED 1

// Set any or all of these switches to enable or disable logging at specific
levels.
#define LOGGING_LEVEL_DEBUG 1
#define LOGGING_LEVEL_INFO 1
#define LOGGING_LEVEL_ERROR 1

// Set this switch to set whether or not to include class, method and line
information in the log entries.
#define LOGGING_INCLUDE_CODE_LOCATION 1

// ***************** END OF USER SETTINGS ***************

#if !(defined(LOGGING_ENABLED) && LOGGING_ENABLED)
#undef LOGGING_LEVEL_DEBUG
#undef LOGGING_LEVEL_INFO
#undef LOGGING_LEVEL_ERROR
```

```
#endif

// Logging format
#define LOG_FORMAT_NO_LOCATION(fmt, lvl, ...) NSLog((@â€□[%@] â€□ fmt), lvl,
##__VA_ARGS__)
#define LOG_FORMAT_WITH_LOCATION(fmt, lvl, ...) NSLog((@â€□%s [Line %d] [%@]
â€□ fmt), __PRETTY_FUNCTION__, __LINE__, lvl, ##__VA_ARGS__)

#if defined(LOGGING_INCLUDE_CODE_LOCATION) && LOGGING_INCLUDE_CODE_LOCATION
#define LOG_FORMAT(fmt, lvl, ...) LOG_FORMAT_WITH_LOCATION(fmt, lvl,
##__VA_ARGS__)
#else
#define LOG_FORMAT(fmt, lvl, ...) LOG_FORMAT_NO_LOCATION(fmt, lvl,
##__VA_ARGS__)
#endif

// Debug level logging
#if defined(LOGGING_LEVEL_DEBUG) && LOGGING_LEVEL_DEBUG
#define LogDebug(fmt, ...) LOG_FORMAT(fmt, @â€□debugâ€□, ##__VA_ARGS__)
#else
#define LogDebug(...)
#endif

// Info level logging
#if defined(LOGGING_LEVEL_INFO) && LOGGING_LEVEL_INFO
#define LogInfo(fmt, ...) LOG_FORMAT(fmt, @â€□infoâ€□, ##__VA_ARGS__)
#else
#define LogInfo(...)
#endif

// Error level logging
#if defined(LOGGING_LEVEL_ERROR) && LOGGING_LEVEL_ERROR
#define LogError(fmt, ...) LOG_FORMAT(fmt, @â€□***ERROR***â€□, ##__VA_ARGS__)
#else
#define LogError(...)
#endif
//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////
```

Appendix D – RDA Matlab code

```
function ExrEchoToRDA()
clear

% INITIALIZATION

% Load the echo data from the EXR file
filename = uigetfile;
[echoData, settings] = exr_rd_echo(filename);

numEchos = size(echoData, 1); % Number of pulse (Azimuth) samples
numEchos = numEchos - mod(numEchos, 2); % make the number of samples even.
rbins=size(echoData, 2); % Number of time (Range) samples

% Main SAR Parameters
PRF=settings(8); % Pulse Repetition Frequency (Hz)
dur=numEchos / PRF; % Time of Flight (sec), PRF*dur = received echoes
vp=settings(8) * settings(10) * settings(9); % Velocity of platform
fo=settings(3); % Carrier frequency
La=settings(7); % Antenna length actual
Xc=settings(1); % Range distance to center of target area
X0=settings(2); % Range width around the center
Xmin = Xc - (X0/2); % Minimum Range
Xmax = Xc + (X0/2); % Maximum Range
Tp=settings(5); % Chirp Pulse Duration
B0=settings(4); % Baseband bandwidth is plus/minus B0

% General Variables
cj=sqrt(-1);
c=3e8; % Propagation speed
ic=1/c; % Propagation frequency
lambda=c/fo; % Wavelength (6cm for fo = 4.5e9)
eta=linspace(0,dur,numEchos)'; % Slow Time Array

% Range Parameters
Kr=B0/Tp; % Range Chirp Rate
dt=1/(2*B0); % Time Domain Sampling Interval
Ts=(2*(Xmin))/c; % Start time of sampling
Tf=(2*(Xmax))/c+Tp; % End time of sampling

% Azimuth Parameters
Ka=(2*vp^2)./(lambda*Xc); % Linear Azimuth FM rate

% Measurement Parameters
t=Ts+(0:rbins-1)*dt; % Time array for data acquisition
s=echoData; % Echoed signal array


% RANGE DOPLER ALGORITHM (RDA)
% Range Reference Signal
td0=t-2*(Xc/c);
```

```
pha20=pi*Kr*((td0.^2)-td0*Tp);
s0=exp(cj*pha20).*(td0 >= 0 & td0 <= Tp);
fs0=fty(s0); % Reference Signal in frequency domain

% Power equalization
amp_max=1/sqrt(2); % Maximum amplitude for equalization
afsb0=abs(fs0);
P_max=max(afsb0);
I=find(afsb0 >= amp_max*P_max);
fs0(I)=((amp_max*(P_max^2)*ones(1,length(I)))./afsb0(I))...
    .*exp(cj*angle(fs0(I)));
deltaR=(lambda^2*(Xc).*(Ka*(dur*0.5-eta)).^2)/(8*vp^2); % RCM
cells=round(deltaR/.56); % .56 meters/cell in range direction
fs=zeros(numEchos,rbins); fsm=fs; fsmb=fs; smb=fs; fsac=fs; sac=fs;

% Range Compression
for k=1:(numEchos);
    fs(k,:)=fty(s(k,:));
    fsm(k,:)=fs(k,:).*conj(fs0);
    smb(k,:)=ifty(fsm(k,:));
end;

% Plot Range Compression Results
figure(2), imagesc(abs(smb))
xlabel('Range, samples'), ylabel('Azimuth, samples')


% Azimuth Reference Signal
smb0=exp(cj*pi*Ka.*eta.*(2*eta(numEchos/2+1)-eta));
fsmb0=ftx(smb0); % Azimuth Matched Filter Spectrum
for l=1:rbins;
    fsmb(:,l)=ftx(smb(:,l)); % Azimuth Fourier Transform
end;

% Range Cell Migration Correction (RCMC)
for k=1:numEchos/2;
    for m=1:rbins-9
        fsmb(k,m)=fsmb(k,m+cells(k));
        fsmb(numEchos-k,m)=fsmb(numEchos-k,m+cells(k));
    end
end;

% Azimuth Compression
for l=1:rbins;
    fsac(:,l)=fsmb(:,l).*conj(fsmb0); % Azimuth Matched Filtering
    sac(:,l)=iftx(fsac(:,l)); % Final Target Image
end;

% Plot Final Results
figure(1), imagesc(abs(sac))
xlabel('Range, samples'), ylabel('Azimuth, samples')

* MATLAB MEX function for reading echo data from EXR file.
```

```
*/

#include "ImathBox.h"
#include "ImfInputFile.h"
#include "ImfArray.h"
#include "ImfChannelList.h"
#include "ImfPixelType.h"
#include "ImfStandardAttributes.h"
#include "Iex.h"

#include "mex.h"

using namespace Imf;
using namespace Imath;
using namespace Iex;

#define SETTING_RANGE_CENTER_OFFSET                  0
#define SETTING_RANGE_WIDTH_OFFSET                   1
#define SETTING_CARRIER_FREQ_OFFSET                  2
#define SETTING_BASEBAND_BW_OFFSET                   3
#define SETTING_PULSE_DURATION_OFFSET                4
#define SETTING_RANGE_SCALE_OFFSET                   5
#define SETTING_ANTENNA_LENGTH_OFFSET                6
#define SETTING_PULSE_REP_FREQ_OFFSET                7
#define SETTING_PIXEL_SPACING_OFFSET                 8
#define SETTING_PIX_PER_PULSE_OFFSET                 9


#define SETTING_TOTAL_NUMBER                         10



/* Check inputs
 * one input: string (row vector of chars)
 *
 * 3 outputs:
 *    float array of real echo portions
 *    float array of imaginary echo portions
 *    float array of settings
 *
 * mexErrMsgTxt returns control directly to Matlab without executing any more
of
 * the C code.  It is not necessary to perform an error return, and check data.
 */
void checkInputs(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[]) {

      if (nrhs != 1)
            mexErrMsgTxt("Incorrect number of input arguments.");

      if (nlhs != 2)
            mexErrMsgTxt("Incorrect number of output arguments.");

      if (mxIsChar(prhs[0]) != 1)
            mexErrMsgTxt("Input must be a string.");
```

```
      if (mxGetM(prhs[0]) != 1)
            mexErrMsgTxt("Input must be a row vector.");

      return;
}

/* Read the settings attributes from the file
 *
 * function is called during a try/catch block.  throwing the errors
 * will invoke the catch routine, and exit the function gracefully.
 */
void ReadAttributes(InputFile &file, double *settingsArray){

      const FloatAttribute *rangeCen = file.header().findTypedAttribute
                                                      <FloatAttribute> ("range
center");
      if (rangeCen == NULL){
            throw "Min Range attribute not found";
      }
      settingsArray[SETTING_RANGE_CENTER_OFFSET] = rangeCen->value();

      const FloatAttribute *rangeWidth = file.header().findTypedAttribute
                                                      <FloatAttribute> ("range
width");
      if (rangeWidth == NULL){
            throw "Range Width attribute not found";
      }
      settingsArray[SETTING_RANGE_WIDTH_OFFSET] = rangeWidth->value();

      const FloatAttribute *frequency = file.header().findTypedAttribute
                                                      <FloatAttribute>
("frequency");
      if (frequency == NULL){
            throw "Frequency attribute not found";
      }
      settingsArray[SETTING_CARRIER_FREQ_OFFSET] = frequency->value();

      const FloatAttribute *baseBw = file.header().findTypedAttribute
                                                      <FloatAttribute>
("baseband bw");
      if (baseBw == NULL){
            throw "Baseband Bandwidth attribute not found";
      }
      settingsArray[SETTING_BASEBAND_BW_OFFSET] = baseBw->value();

      const FloatAttribute *pulseDur = file.header().findTypedAttribute
                                                      <FloatAttribute> ("pulse
duration");
      if (pulseDur == NULL){
            throw "Pulse Duration attribute not found";
      }
      settingsArray[SETTING_PULSE_DURATION_OFFSET] = pulseDur->value();

      const FloatAttribute *rangeScale = file.header().findTypedAttribute
                                                      <FloatAttribute> ("range
```

```
scale");
      if (rangeScale == NULL){
            throw "Range Scale attribute not found";
      }
      settingsArray[SETTING_RANGE_SCALE_OFFSET] = rangeScale->value();

      const FloatAttribute *antLen = file.header().findTypedAttribute
                                                      <FloatAttribute>
("antenna length");
      if (antLen == NULL){
            throw "Antenna Length attribute not found";
      }
      settingsArray[SETTING_ANTENNA_LENGTH_OFFSET] = antLen->value();

      const FloatAttribute *pulseRep = file.header().findTypedAttribute
                                                      <FloatAttribute> ("pulse
repetition freq");
      if (pulseRep == NULL){
            throw "Pulse Repetition Frequency attribute not found";
      }
      settingsArray[SETTING_PULSE_REP_FREQ_OFFSET] = pulseRep->value();

      const FloatAttribute *pulseSpace = file.header().findTypedAttribute
                                                      <FloatAttribute> ("pixel
spacing");
      if (pulseSpace == NULL){
            throw "Pixel Spacing attribute not found";
      }
      settingsArray[SETTING_PIXEL_SPACING_OFFSET] = pulseSpace->value();

      const FloatAttribute *pixPerPulse = file.header().findTypedAttribute
                                                      <FloatAttribute> ("pixels
per pulse");
      if (pixPerPulse == NULL){
            throw "Pixels Per Pulse attribute not found";
      }
      settingsArray[SETTING_PIX_PER_PULSE_OFFSET] = pixPerPulse->value();

}


/* Main function called from Matlab.
 *
 * First checks to make sure the function was called correctly,
 * then calls a function to load the settings that were saved in the attributes
of the files.
 * finally reads the data from the file (real in Red "R" channel, imaginary in
Green "G"
 * channel, and copies it into the Matlab array.
 */
void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[]) {

      checkInputs(nlhs, plhs, nrhs, prhs);
      char *filename = mxArrayToString(prhs[0]);
```

```
      try {
            InputFile file(filename);
            mxFree(filename);

            // read the settings from the echo file.
            int dims[2] = {SETTING_TOTAL_NUMBER, 1};
            plhs[1] = mxCreateNumericArray(2, dims, mxDOUBLE_CLASS, mxREAL);
            double *settings = mxGetPr(plhs[1]);
            ReadAttributes(file, settings);

            // get the image dimensions
            Box2i dw = file.header().dataWindow();

            int x  = dw.max.x - dw.min.x + 1;
            int y = dw.max.y - dw.min.y + 1;

            // allocate the buffers for the floating point data from the file
            Array2D<float> re(y,x);
            Array2D<float> im(y,x);

            // add the buffers to the frame buffer construct to prepare for
reading
            FrameBuffer frmBuf;
            frmBuf.insert("R", Slice(FLOAT, (char *)&re[0][0], sizeof(float),
sizeof(float) * x, 1, 1, 0.0));
            frmBuf.insert("G", Slice(FLOAT, (char *)&im[0][0], sizeof(float),
sizeof(float) * x, 1, 1, 0.0));

            // read the data from the file
            file.setFrameBuffer(frmBuf);
            file.readPixels(dw.min.y, dw.max.y);

            // allocate the Matlab variable space
            dims[0] = y;
            dims[1] = x;
            plhs[0] = mxCreateNumericArray(2, dims, mxDOUBLE_CLASS, mxCOMPLEX);
            double *echoReal = mxGetPr(plhs[0]);
            double *echoImaginary = mxGetPi(plhs[0]);

            // copy the data from the "C" constructs to the Matlab constructs.
            for (int i = 0; i < y; ++i) {
                  for (int j = 0; j < x; ++j) {
                        int k = j*y + i;

                        echoReal[k]             = re[i][j];
                        echoImaginary[k]  = im[i][j];
                  }
            }

      // if anything goes wrong from the try { this next }, this
      // block of code will get called, clean up allocated memory, and
      // print an error message in the Matlab window.
      } catch (const std::exception &exc) {
            mxFree(filename);
            mexErrMsgTxt(exc.what());
```

```
        }

        return;
}
```

Appendix E – Code for Original Approach (900 images processed)

1. ANTENNA PATTERN CODE

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
*
 *  PatternGenAndSave.cpp
 *  AntennaPatternGen
 *
 *
 */

#include "PatternGenAndSave.h"
#include <cmath>

using namespace std;

#define PI 3.14159265358979323846
#define PLANE_DIST 20000.0

double sinc(double th){
	if(th == 0)
		return 1.0;
	return sin(th)/th;
}

// This pattern generation function implements a sinc squared function
// Wa = sinc(0.866 * theta / BW)^2
// BW = 0.866lamda/antennaLength
// thus the final sinc(theta * antLenth/waveLength)^2.
// added to this pattern generation is now also a range adjustment.
// Blender output a distance from the plane of the camera to the object.
// To get the range from the point of the camera, the range needs to be modified
// by the inverse cosine of the angle.
void GenPattern(float *data, float *rangeMod, int width, int height, float angle,
				float antLenth, float waveLength, float rangeScale){
	int i, j;
	int x, y;

	double xAngle;
	double yAngle;
	double singlePixelAngle = angle / (width + 1);
	double radialAngle;

	double xDisplacement, yDisplacement;
	double oppSide;

	// calculate this outside the loop to save loop execution time.
	double syncMod	= antLenth/waveLength;

	// convert to radians for the math functions
	singlePixelAngle = singlePixelAngle * PI / 180.0;

	for (j=0; j<height; j++) {
```

```
            y = j - height/2;
            yAngle = abs(y) * singlePixelAngle;
            yDisplacement = tan(yAngle) * PLANE_DIST;
            for (i = 0; i < width; i++) {
                  x = i - width/2;
                  xAngle = abs(x) * singlePixelAngle;
                  xDisplacement = tan(xAngle) * PLANE_DIST;

                  oppSide = sqrt(pow(xDisplacement, 2.0) + pow(yDisplacement, 2.0));
                  radialAngle = atan2(oppSide, PLANE_DIST);
                  double sincRes = sinc(radialAngle * syncMod);
                  // square the sinc result and store in the array.
                  data[i + (j * width)] = pow(sincRes, 2.0);
                  rangeMod[i + (j*width)] = rangeScale/cos(radialAngle);
            }
      }
}

void saveEXR(const char* filename, int width, int height, const float* data, const float*
rangeMod,
                   float angle, float frequency, float antLen, float rangeScale){
      Header header (width, height);
      header.insert("hrzangle", FloatAttribute (angle));
      header.insert("frequency", FloatAttribute (frequency));
      header.insert("antenna length", FloatAttribute (antLen));
      header.insert("range scale", FloatAttribute (rangeScale));
      header.channels().insert ("G", Channel (FLOAT));
      header.channels().insert ("Z", Channel (FLOAT));

      OutputFile file (filename, header);

      FrameBuffer frameBuffer;

      frameBuffer.insert ("G", Slice(FLOAT, (char *)data, sizeof(*data),
sizeof(*data)*width));
      frameBuffer.insert ("Z", Slice(FLOAT, (char *)rangeMod, sizeof(*rangeMod),
                                                sizeof(*rangeMod)*width));

      file.setFrameBuffer(frameBuffer);
      file.writePixels(height);
}
////////////////////////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////////////////////////
/
//  PatternView.h
//  AntennaPatternGen
//
//

#import <Cocoa/Cocoa.h>

@interface PatternView : NSView {

      int imgwidth, imgheight;
```

```
        float imgangle;
        float antLength;
        float waveLength;
        float imgexposure;
        float frequency;
        float rangeScale;

        float *data;
        float *rangeMod;

        NSBitmapImageRep *myImageCache;
        unsigned char *myDataPtr;

        IBOutlet id HeightSetting;
        IBOutlet id WidthSetting;
        IBOutlet id AngleSetting;
        IBOutlet id AntennaLengthSetting;
        IBOutlet id FrequencySetting;
        IBOutlet id ExposureSetting;
        IBOutlet id RangeScaleSetting;
        IBOutlet NSWindow* progWin;

}

- (void)awakeFromNib ;
- (void) drawRect:(NSRect)dirtyRect ;
- (IBAction) SettingChange:
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//
//  AntennaPatternGenAppDelegate.h
//  AntennaPatternGen
//
//

#import <Cocoa/Cocoa.h>

@interface AntennaPatternGenAppDelegate : NSObject <NSApplicationDelegate> {
    NSWindow *window;
}

@property (assign) IBOutlet NSWindow *window;

@end
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

/*
 *  PatternGenAndSave.h
 *  AntennaPatternGen
 *
 *
```

```
 */

#ifndef PATTERNGENANDSAVE_H
#define PATTERNGENANDSAVE_H

#ifdef __cplusplus

#include <ImfRgbaFile.h>
#include <ImfChannelList.h>
#include <ImfHeader.h>
#include <ImfOutputFile.h>
#include <ImfInputFile.h>
#include <ImfStandardAttributes.h>
#include <ImfArray.h>

using namespace Imf;
using namespace Imath;

#endif

#ifdef __cplusplus
extern "C" {
#endif

       void GenPattern(float *data, float *rangeMod, int width, int height, float angle,
                                    float antLength, float waveLength, float rangeScale);
       void saveEXR(const char* filename, int width, int height, const float* data,
                            const float* rangeMod, float angle, float frequency, float
antLen,
                            float rangeScale);

#ifdef __cplusplus
}
#endif

#endif //PATTERNGENANDSAVE_H

////////////////////////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////////////////////////

//
// Prefix header for all source files of the 'AntennaPatternGen' target in the
'AntennaPatternGen' project
//

#ifdef __OBJC__
    #import <Cocoa/Cocoa.h>
#endif

////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////

//
```

```
//  AntennaPatternGenAppDelegate.m
//  AntennaPatternGen
//
//

#import "AntennaPatternGenAppDelegate.h"

@implementation AntennaPatternGenAppDelegate

@synthesize window;

- (void)applicationDidFinishLaunching:(NSNotification *)aNotification {
      // Insert code here to initialize your application
}

@end

////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////

//
//  main.m
//  AntennaPatternGen
//
//

#import <Cocoa/Cocoa.h>

int main(int argc, char *argv[])
{
    return NSApplicationMain(argc,  (const char **) argv);
}

////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////

//
//  PatternView.m
//  AntennaPatternGen
//
//

#import "PatternView.h"

#import "PatternGenAndSave.h"


@implementation PatternView

- (void)awakeFromNib
{
      imgexposure = 1.f;
      imgwidth = 2880;
```

```
      imgheight = 2880;
      imgangle = 7.32f;
      antLength = 2.f;
      frequency = 4.5e9f;
      waveLength = 3e8/frequency;
      rangeScale = 10.0;
}

- (void) SettingChange: (id)Sender {
      imgheight = [HeightSetting intValue];
      imgwidth = [WidthSetting intValue];
      imgangle = [AngleSetting floatValue];
      antLength = [AntennaLengthSetting floatValue];
      waveLength = 3e8/[FrequencySetting floatValue];
      rangeScale = [RangeScaleSetting floatValue];

      if(!data)
            data = (float *)malloc(imgwidth*imgheight*sizeof(float));
      if(!rangeMod)
            rangeMod = (float *)malloc(imgwidth*imgheight*sizeof(float));

      GenPattern(data, rangeMod, imgwidth, imgheight, imgangle, antLength,
waveLength, rangeScale);

      [self setNeedsDisplay:YES];
}

- (void) SizeChange: (id)Sender {
      imgheight = [HeightSetting intValue];
      imgwidth = [WidthSetting intValue];

      if(data) free(data);
      data = (float *)malloc(imgwidth*imgheight*sizeof(float));
      if(rangeMod) free(rangeMod);
      rangeMod = (float *)malloc(imgwidth*imgheight*sizeof(float));

      GenPattern(data, rangeMod, imgwidth, imgheight, imgangle,
antLength,waveLength, rangeScale);

      [self setNeedsDisplay:YES];
}

- (void) ExposureChange: (id)Sender {
      imgexposure = pow(2,[ExposureSetting floatValue]);

      [self setNeedsDisplay:YES];
}


- (void) drawRect:(NSRect)dirtyRect {
      NSRect bds = dirtyRect;

      int myRowBytes = 4 * bds.size.width;

      if(myDataPtr) free(myDataPtr);
```

```
      if(myImageCache) [myImageCache release];

      myDataPtr = (unsigned char *)malloc(myRowBytes * bds.size.height);
      myImageCache = [[NSBitmapImageRep alloc]
initWithBitmapDataPlanes:&myDataPtr

         pixelsWide:bds.size.width

         pixelsHigh:bds.size.height

      bitsPerSample:8

samplesPerPixel:4

             hasAlpha:YES

             isPlanar:NO

colorSpaceName:NSCalibratedRGBColorSpace

       bitmapFormat:0

        bytesPerRow:4*bds.size.width

       bitsPerPixel:32];

      if(data) {
            int i, j;
            float x, y, cap;

            for (j=0; j<bds.size.height; j++) {

                  y = (float)j/bds.size.height;

                  for (i=0; i<bds.size.width; i++) {

                        x = (float)i/bds.size.width;

                        int drawIndex = j*bds.size.width + i;
                        int imgx = x*imgwidth;
                        int imgy = y*imgheight;
                        int imgIndex = imgy*imgwidth + imgx;

                        cap = data[imgIndex]*imgexposure;
                        if(cap > 1) cap = 1;
                        myDataPtr[drawIndex*4] = cap*255;
                        myDataPtr[drawIndex*4+1] = cap*255;
                        cap = rangeMod[imgIndex]*imgexposure;
                        if(cap > 1) cap = 1;
                        myDataPtr[drawIndex*4+2] = cap*255;
                        myDataPtr[drawIndex*4+3] = 255;
                  }
            }

      }
```

```
      [myImageCache draw];

}

- (void) SaveImage: (id)Sender {
      NSLog(@"doSaveAs");
      NSSavePanel *tvarNSSavePanelObj = [NSSavePanel savePanel];
      int tvarInt = [tvarNSSavePanelObj runModal];
      if(tvarInt == NSOKButton){
            NSLog(@"doSaveAs we have an OK button");
      } else if(tvarInt == NSCancelButton) {
            NSLog(@"doSaveAs we have a Cancel button");
            return;
      } else {
            NSLog(@"doSaveAs tvarInt not equal 1 or zero = %3d",tvarInt);
            return;
      } // end if
      NSString * tvarDirectory = [tvarNSSavePanelObj directory];
      NSLog(@"doSaveAs directory = %@",tvarDirectory);
      NSString * tvarFilename = [tvarNSSavePanelObj filename];
      NSLog(@"doSaveAs filename = %@",tvarFilename);

      saveEXR([tvarFilename UTF8String], imgwidth, imgheight, data, rangeMod,
                  imgangle, frequency, antLength, rangeScale);

}

@end
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

2. ECHO GENERATOR CODE

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

/*
 *  imagetoecho.cpp
 *  EchoGenerator
 *
 *
 */

#include "imagetoecho.h"
#include "logging.h"

#include <complex>


#define PI 3.14159265358979323846264338327950

/* function to take details about the transmitted SAR pulse, and calculate the output of
 * the quadrature demodulation circuit based on an ideal return.
 * The setup structure contains data about the simulation parameters.
 * The signal array pointer is a pointer to an array of the size of the numQuadDemodSamples
```

```
 * element of the setup structure.
 * This value must be greater than or equal to the value defined by the
 * length of the transmitted pulse multiplied by the sampling frequency.  If the array
length
 * is not greater than this value, this function will return -1;
 * The arrayLen value states the length of the array pointed to by signal.
 */
int QuadDemodXmitSignalCalculate(IMAGE_TO_ECHO_SETUP *setup, std::complex<float> *signal){
	float pulseDur = setup->pulseDuration;
	// The signal chirp rate is determined by the desired bandwidth, and how long the
pulse is
	// transmitted.
	float chirpRate = setup->basebandBW / pulseDur;
	// To properly sample the data from the quadrature demodulation, there needs to be
as many
	// samples as the frequency range.  The baseband bandwidth is a +/- from center, so
	// multiply by 2.
	float sampleTime = 1.0 / (2.0 * setup->basebandBW);
	// The number of samples on the ideal reflection is determined by the number of
	// samples that occur during the transmitted pulse width.
	int numSamples = (int)(pulseDur / sampleTime);

	// i * PI * chirpRate is a portion of the exponential calculation for the quad demod
	// function.  By precalculating it outside the loop, we reduce execution time.
	std::complex<float> piKr = std::complex<float>(0.0, PI * chirpRate);

	float timeOffset;

	int arrayLen = setup->numQuadDemodSamples;

	for (int i=0; i < arrayLen; i++) {
		if (i<numSamples) {
			timeOffset = i * sampleTime;
			// calculate the value
			signal[i] = exp(piKr * ((float)pow(timeOffset, 2.0) - (timeOffset *
pulseDur)));
		} else {
			signal[i] = std::complex<float>(0.0, 0.0);
		}

	}
	// check to inform the calling function that the array was not large enough to hold
	// the entire signal demodulation signature.
	if (arrayLen < numSamples){
//		LogDebug(@"arrayLen (%d) is smaller than the number of samples (%d).",
//				arrayLen, numSamples);
		return -1;
	}

	return 0;
}

/* function to take an antenna radiance pattern, an image with reflectance information
 * and a set of range information, and a demodulation signature, and create an echo.
 * The antenna radiance pattern, the image, and the range data are two dimensional arrays
 * of the same size, defined by the width and height fields of the setup structure.
```

```
 * The quadDemodSignal array is a one dimensional array containing the demodulation
 * signature of the transmitted pulse.  This array size is contained in the
numQuadDemodSamples
 * field of the setup data.  The array is multiplied by target location/reflectance as part
 * of creating the echo signal.
 * The echo array is a one dimensional array of the size defined by the numEchoBins field
 * of the setup structure.
 */
int ImageToEcho(IMAGE_TO_ECHO_SETUP *setup, float *antennaPattern, float *rangeMod, float
*image,
                            float *range, std::complex<float> *quadDemodSignal,
std::complex<float> *echo){
       int numPix = setup->height * setup->width;

       int reflectLen = setup->numEchoBins + setup->numQuadDemodSamples;

       // The reflection timeline must contain reflections that start before the
       // receiver if the reflection itself would continue into the area where the receiver
       // is active.  Any signal with this characteristic acts as a noise input into the
       // start of matched filtering.
       std::complex<float> reflection[reflectLen];
       std::complex<float> zero(0.0, 0.0);

       // the rangeTime variable here is the distance light travels for each sample in the
       // range direction.  The calculation calculates the inverse, so we can multiply in
the loop.
       // Multiplication is generally less computationally intensive than division.
       // The sampling time is multiplied by 2 in order to take into account that the light
goes
       // out and back, thus needing to cover 2X the range that we are measuring.
       float rangeTime = 2.0 * 2.0 * setup->basebandBW / SPEED_OF_LIGHT;

       std::complex<float> reflectMagnitude;
       int index;

       // get the ranges where the receiver gets data.  Read comment above reflection for
an
       // explanation of why the min range is modified.
//     float rangeScale = setup->rangeScale;
//     float minRange = (setup->minRange - (setup->numQuadDemodSamples /
rangeTime))/rangeScale;
//     float maxRange = setup->maxRange/rangeScale;
//     float rangeScale = setup->rangeScale;
       float minRange = (setup->minRange - (setup->numQuadDemodSamples / rangeTime));
       float maxRange = setup->maxRange;

       std::complex<float> fourPiF0DivByC = std::complex<float>(0.0,
                                                                -4.0 * PI *
(float)setup->carrierFreq / SPEED_OF_LIGHT);
       for (int i=0; i<numPix; i++) {
              // check if the target is within the range that the receiver is receiving
data
              float curRange = range[i] * rangeMod[i];
              if ((curRange > minRange) && (curRange < maxRange)) {
                     if((image[i] > 1.0f) || (image[i] < 0.0f)){
//                          LogError(@"image pixel %d has a value (%f) outside limits", i,
```

```
image[i]);
                        return -2;
                    }
                    // calculate the magnitude of the reflection
                    reflectMagnitude = (float)(antennaPattern[i] * image[i]) *
                                                                    exp(fourPiF0DivByC *
(float)(curRange));
                    // determine when the reflection happens on the time line
                    index = round((curRange - minRange) * rangeTime);
                    if (index > reflectLen) {
//                      LogError(@"index (%d) ended up longer than reflectLen (%d)!",
index, reflectLen);
                        return -1;
                    } else {
                        // add that reflection to the bin.
                        reflection[index] += reflectMagnitude;
                    }
                    break;
                }
        }

        int sigLen = setup->numQuadDemodSamples;

        // multiply the reflection data by the demodulation signature
        // check each location
        for (int location = 0; location < reflectLen; location++){
            if(reflection[location] != zero){
                    // and multiply by each signature location
                    for (int sigLoc = 0; sigLoc < sigLen; sigLoc++){
                        int echoLoc = sigLoc + location;
                        // calculate if the signature+location places the
multiplication value within
                        // the receiver window.  Part of the location information is
previous to the
                        // window as mentioned in the reflection comment, and part
extends beyond the window.
                        if ((echoLoc < reflectLen) && (echoLoc >= sigLen)){
                            // add the echo signature into the final echo value.
multiple signals will overlap
                            // into the echo locations, so the value must be added
to the signal already in the bin.
                            echo[echoLoc - sigLen] += reflection[location] *
quadDemodSignal[sigLoc];
                        }
                    }
                }
        }
        return 0;
}
////////////////////////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////////////////////////

/*
 *  loadimage.cpp
```

```
 *  EchoGenerator
 *
 */

#include "loadimage.h"
//#include "logging.h"

#include <ImfRgbaFile.h>
#include <ImfChannelList.h>
#include <ImfHeader.h>
#include <ImfOutputFile.h>
#include <ImfInputFile.h>
#include <ImfStandardAttributes.h>
#include <ImfArray.h>

using namespace Imf;
using namespace Imath;

/* function to get the width and height of an OpenEXR image.
 */
int GetImageSize(const char *filename, int *width, int *height){
	InputFile file(filename);
	Box2i dw = file.header().dataWindow();
	*width  = dw.max.x - dw.min.x + 1;
	*height = dw.max.y - dw.min.y + 1;

	return 0;
}


/* function loads the antenna pattern from a file into an array.
 * returns an error if width and height are not equal to the image header data.
 */
int LoadAntennaPattern(const char *filename, int width, int height, float *data, float
*rangeMod,
                                        float *carrier, float *patternAngle, float
*antLength, float *rangeScale){
	// open the file, and get the internal window size structure.
	InputFile file(filename);
	Box2i dw = file.header().dataWindow();

	// check the size of the file versus the arguments passed into the function
	// return an error if they don't match.
	if ((dw.max.x - dw.min.x + 1 != width) || (dw.max.y - dw.min.y + 1 != height)) {
		//		LogError(@"image height and width don't match given
dimensions.");
		return -1;
	}

	const FloatAttribute *frequency = file.header().findTypedAttribute

	<FloatAttribute> ("frequency");
	if (frequency == NULL){
//		LogError(@"frequency attribute not found in the file");
		return -2;
	}
```

```
	*carrier = frequency->value();

	const FloatAttribute *angle = file.header().findTypedAttribute

	<FloatAttribute> ("hrzangle");
	if (angle == NULL){
		//		LogError(@"Horizontal Angle attribute not found in the file");
		return -3;
	}

	*patternAngle = angle->value();

	const FloatAttribute *antennaLen = file.header().findTypedAttribute

	<FloatAttribute> ("antenna length");
	if (antennaLen == NULL){
		//		LogError(@"antenna length attribute not found in the file");
		return -4;
	}

	*antLength = antennaLen->value();

	const FloatAttribute *rangeScaleAttr = file.header().findTypedAttribute

	<FloatAttribute> ("range scale");
	if (rangeScaleAttr == NULL){
		//		LogError(@"range scale attribute not found in the file");
		return -5;
	}

	*rangeScale = rangeScaleAttr->value();

	// create the framebuffer class for IMF
	FrameBuffer frameBuffer;
	frameBuffer.insert ("G",				// name
				Slice (FLOAT,				// type
					(char *) data,
					sizeof (*data) * 1,    // xStride
					sizeof (*data) * (width),// yStride
					1, 1,				// x/y
sampling
					0.0));				//
fillValue

	frameBuffer.insert ("Z",				// name
				Slice (FLOAT,				// type
					(char *) rangeMod,
					sizeof (*rangeMod) * 1,    // xStride
					sizeof (*rangeMod) * (width),// yStride
					1, 1,				// x/y
sampling
					0.0));				//
fillValue

	file.setFrameBuffer (frameBuffer);
```

```
	// read the data from the file
	file.readPixels (dw.min.y, dw.max.y);

	return 0;
}


/* function loads image data from an OpenEXR file.
 * loads RGB data, and sets to zero if R, G, and B are not the same.
 * loads the Z data into the range buffer.
 * function returns an error if width and height are not equal to the image header width
 * and height.
 */
int LoadImageData(const char *filename, int width, int height, float *data, float *range){
	// open the file, and get the internal window size structure.
	InputFile file(filename);
	Box2i dw = file.header().dataWindow();

	// check the size of the file versus the arguments passed into the function
	// return an error if they don't match.
	if ((dw.max.x - dw.min.x + 1 != width) || (dw.max.y - dw.min.y + 1 != height)) {
//		LogError(@"image height and width don't match given dimensions.");
		return -1;
	}

	// create the framebuffer class for IMF
	FrameBuffer frameBuffer;
	frameBuffer.insert ("G",                                    // name
                                    Slice (FLOAT,                           // type
                                           (char *) data,
                                           sizeof (data[0]) * 1,    // xStride
                                           sizeof (data[0]) * (width),// yStride
                                           1, 1,                            // x/y
sampling
                                           0.0));                           //
fillValue

	frameBuffer.insert ("Z",                                    // name
                                    Slice (FLOAT,                           // type
                                           (char *) range,
                                           sizeof (range[0]) * 1,    // xStride
                                           sizeof (range[0]) * (width),// yStride
                                           1, 1,                            // x/y
sampling
                                           0.0));                           //
fillValue

	file.setFrameBuffer (frameBuffer);
	// read the data from the file
	file.readPixels (dw.min.y, dw.max.y);

	return 0;

}

/* function saves echo data to an OpenEXR file.
```

```
 * saves data as floating point values, real in channel "R", and imaginary in channel "G"
 * settings get included in the EXR header as float attributes.  The name of the float
 * attribute is based on the constant names defined in the header file.
 */
int SaveEchoData(const char *filename, int width, int height, std::complex<float> *data,
                            float *settings){

       Header header (width, height);
       if(settings != NULL){
              header.insert("min range", FloatAttribute
(settings[SETTING_MIN_RANGE_OFFSET]));
              header.insert("max range", FloatAttribute
(settings[SETTING_MAX_RANGE_OFFSET]));
              header.insert("frequency", FloatAttribute
(settings[SETTING_CARRIER_FREQ_OFFSET]));
              header.insert("baseband bw", FloatAttribute
(settings[SETTING_BASEBAND_BW_OFFSET]));
              header.insert("pulse duration", FloatAttribute
(settings[SETTING_PULSE_DURATION_OFFSET]));
              header.insert("range scale", FloatAttribute
(settings[SETTING_RANGE_SCALE_OFFSET]));
              header.insert("antenna length", FloatAttribute
(settings[SETTING_ANTENNA_LENGTH_OFFSET]));
              header.insert("antenna angle", FloatAttribute (

       settings[SETTING_ANTENNA_PATTERN_ANGLE_OFFSET]));
       }
       header.channels().insert ("R", Channel (FLOAT));
       header.channels().insert ("G", Channel (FLOAT));

       float rpix[width*height];
       float gpix[width*height];

       for(int i=0; i<width*height; i++){
              rpix[i] = data[i].real();
              gpix[i] = data[i].imag();
       }

       OutputFile file (filename, header);

       FrameBuffer frameBuffer;

       frameBuffer.insert ("G", Slice(FLOAT, (char *)gpix, sizeof(*gpix),
sizeof(*gpix)*width));
       frameBuffer.insert ("R", Slice(FLOAT, (char *)rpix, sizeof(*rpix),
sizeof(*rpix)*width));

       file.setFrameBuffer(frameBuffer);
       file.writePixels(height);


       return 0;
}
////////////////////////////////////////////////////////////////////////////////////////////////////
```

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////

*
 *  logging.h
 */

/*
 * There are three levels of logging: debug, info and error, and each can be enabled
independently
 * via the LOGGING_LEVEL_DEBUG, LOGGING_LEVEL_INFO, and LOGGING_LEVEL_ERROR switches below,
respectively.
 * In addition, ALL logging can be enabled or disabled via the LOGGING_ENABLED switch
below.
 *
 * To perform logging, use any of the following function calls in your code:
 *
 * LogDebug(fmt, ...) - will print if LOGGING_LEVEL_DEBUG is set on.
 * LogInfo(fmt, ...) - will print if LOGGING_LEVEL_INFO is set on.
 * LogError(fmt, ...) - will print if LOGGING_LEVEL_ERROR is set on.
 *
 * Each logging entry can optionally automatically include class, method and line
information by
 * enabling the LOGGING_INCLUDE_CODE_LOCATION switch.
 *
 * Logging functions are implemented here via macros, so disabling logging, either
entirely,
 * or at a specific level, removes the corresponding log invocations from the compiled
code,
 * thus completely eliminating both the memory and CPU overhead that the logging calls
would add.
 */

// Set this switch to enable or disable ALL logging.
#define LOGGING_ENABLED 1

// Set any or all of these switches to enable or disable logging at specific levels.
#define LOGGING_LEVEL_DEBUG 1
#define LOGGING_LEVEL_INFO 1
#define LOGGING_LEVEL_ERROR 1

// Set this switch to set whether or not to include class, method and line information in
the log entries.
#define LOGGING_INCLUDE_CODE_LOCATION 1

// ***************** END OF USER SETTINGS ***************

#if !(defined(LOGGING_ENABLED) && LOGGING_ENABLED)
#undef LOGGING_LEVEL_DEBUG
#undef LOGGING_LEVEL_INFO
#undef LOGGING_LEVEL_ERROR
#endif

// Logging format
```

```
#define LOG_FORMAT_NO_LOCATION(fmt, lvl, ...) NSLog((@"[%@] " fmt), lvl, ##__VA_ARGS__)
#define LOG_FORMAT_WITH_LOCATION(fmt, lvl, ...) NSLog((@"%s [Line %d] [%@] " fmt),
__PRETTY_FUNCTION__, __LINE__, lvl, ##__VA_ARGS__)

#if defined(LOGGING_INCLUDE_CODE_LOCATION) && LOGGING_INCLUDE_CODE_LOCATION
#define LOG_FORMAT(fmt, lvl, ...) LOG_FORMAT_WITH_LOCATION(fmt, lvl, ##__VA_ARGS__)
#else
#define LOG_FORMAT(fmt, lvl, ...) LOG_FORMAT_NO_LOCATION(fmt, lvl, ##__VA_ARGS__)
#endif

// Debug level logging
#if defined(LOGGING_LEVEL_DEBUG) && LOGGING_LEVEL_DEBUG
#define LogDebug(fmt, ...) LOG_FORMAT(fmt, @"debug", ##__VA_ARGS__)
#else
#define LogDebug(...)
#endif

// Info level logging
#if defined(LOGGING_LEVEL_INFO) && LOGGING_LEVEL_INFO
#define LogInfo(fmt, ...) LOG_FORMAT(fmt, @"info", ##__VA_ARGS__)
#else
#define LogInfo(...)
#endif

// Error level logging
#if defined(LOGGING_LEVEL_ERROR) && LOGGING_LEVEL_ERROR
#define LogError(fmt, ...) LOG_FORMAT(fmt, @"***ERROR***", ##__VA_ARGS__)
#else
#define LogError(...)
#endif
////////////////////////////////////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//
//  EchoViewer.h
//  EchoGenerator
//
//

#import <Cocoa/Cocoa.h>


@interface EchoViewer : NSView {
       NSBitmapImageRep *displayImage;
}

- (void) drawRect:(NSRect)dirtyRect;
- (void) SetDisplayImage: (NSBitmapImageRep *)image;

@end
////////////////////////////////////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

```
//
//  EchoGeneratorAppDelegate.h
//  EchoGenerator
//

#import <Cocoa/Cocoa.h>
#import "imagetoecho.h"
#import "EchoViewer.h"

@interface EchoGeneratorAppDelegate : NSObject <NSApplicationDelegate> {
    NSWindow *window;

       IMAGE_TO_ECHO_SETUP imageToEchoSetup;
       float *antennaPattern;
       float *rangeMod;
       float *imageData;
       float *rangeData;
       std::complex<float> *quadDemodSignal;
       std::complex<float> *echo;
       double imgexposure;
       NSInteger numFiles;

       IBOutlet id minRange;
       IBOutlet id maxRange;
       IBOutlet id carrierFreq;
       IBOutlet id basebandBW;
       IBOutlet id pulseDur;
       IBOutlet id rangeScale;
       IBOutlet id echoBins;
       IBOutlet id demodSamples;
       IBOutlet id height;
       IBOutlet id width;
       IBOutlet id ExposureSetting;
       IBOutlet EchoViewer *echoImage;
}

@property (assign) IBOutlet NSWindow *window;

- (IBAction) EchoSetupChange: (id)Sender;
- (IBAction) LoadAntennaPattern: (id)Sender;
- (IBAction) LoadRangeSamples: (id)Sender;
- (IBAction) SaveEchoData: (id)Sender;
- (IBAction) ExposureChange: (id)Sender;

@end

//////////////////////////////////////////////////////////////////////////////////////////



//////////////////////////////////////////////////////////////////////////////////////////

/*
 *  imagetoecho.h
 *  EchoGenerator
```

```
 *
 *
 */

#ifndef IMAGE_TO_ECHO_H
#define IMAGE_TO_ECHO_H

#define SPEED_OF_LIGHT 299792458.0

#include <complex>

typedef struct {
	int height;
	int width;
	float minRange;
	float maxRange;
	int numEchoBins;
	int numQuadDemodSamples;
	float carrierFreq;
	float basebandBW;
	float pulseDuration;
	float rangeScale;
	float antLength;
	float antPatAngle;
} IMAGE_TO_ECHO_SETUP;

/* function to take details about the transmitted SAR pulse, and calculate the output of
 * the quadrature demodulation circuit based on an ideal return.
 * The setup structure contains data about the simulation parameters.
 * The signal array pointer is a pointer to an array of the size defined by the
 * length of the transmitted pulse multiplied by the sampling frequency.
 */
int QuadDemodXmitSignalCalculate(IMAGE_TO_ECHO_SETUP *setup, std::complex<float> *signal);

/* function to take an antenna radiance pattern, an image with reflectance information,
 * and a set of range information, and create an echo.
 * The antenna radiance pattern, the image, and the range data are two dimensional arrays
 * of the same size, defined by the width and height fields of the setup structure.
 * The echo array is a one dimensional array of the size defined by the numEchoBins field
 * of the setup structure.
 * This function does not allocate memory for any of the arguments.  They must have been
 * allocated outside this function and passed in.  It will also not free any of these
 * arguments.
 * For internal use, the function will allocate a buffer the size of the image, and delete
 * it before returning.
 */
int ImageToEcho(IMAGE_TO_ECHO_SETUP *setup, float *antennaPattern, float *rangeMod, float
*image,
                            float *range, std::complex<float> *quadDemodSignal,
std::complex<float> *echo);


#endif // IMAGE_TO_ECHO_H
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////

/*
 *  loadimage.h
 *  EchoGenerator
 *
 */

#ifndef LOAD_IMAGE_H
#define LOAD_IMAGE_H

#include <complex>

#define SETTING_MIN_RANGE_OFFSET                        0
#define SETTING_MAX_RANGE_OFFSET                        1
#define SETTING_CARRIER_FREQ_OFFSET                             2
#define SETTING_BASEBAND_BW_OFFSET                      3
#define SETTING_PULSE_DURATION_OFFSET                   4
#define SETTING_RANGE_SCALE_OFFSET                      5
#define SETTING_ANTENNA_LENGTH_OFFSET                   6
#define SETTING_ANTENNA_PATTERN_ANGLE_OFFSET    7

#define SETTING_TOTAL_NUMBER                                    8


/* function to get the width and height of an OpenEXR image.
 */
int GetImageSize(const char *filename, int *width, int *height);

/* function loads the antenna pattern from a file into an array.
 * returns an error if width and height are not equal to the image header data.
 */
int LoadAntennaPattern(const char *filename, int width, int height, float *data, float
*rangeMod,
                                        float *carrierFreq, float *patternAngle, float
*antLength, float *rangeScale);

/* function loads image data from an OpenEXR file.
 * loads RGB data, and sets to zero if R, G, and B are not the same.
 * loads the Z data into the range buffer.
 * function returns an error if width and height are not equal to the image header width
 * and height.
 */
int LoadImageData(const char *filename, int width, int height, float *data, float *range);

/* function saves echo data to an OpenEXR file.
 * saves data as floating point values, real in channel "R", and imaginary in channel "G"
 * settings get included in the EXR header as float attributes.  The name of the float
 * attribute is based on the constant names defined in the header file.
 */
int SaveEchoData(const char *filename, int width, int height, std::complex<float> *data,
                             float *settings);
```

```
#endif //LOAD_IMAGE_H
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////



////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//
//  main.m
//  EchoGenerator
//
//

#import <Cocoa/Cocoa.h>

int main(int argc, char *argv[])
{
    return NSApplicationMain(argc,  (const char **) argv);

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////



////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//
//  EchoViewer.m
//  EchoGenerator
//
//

#import "EchoViewer.h"


@implementation EchoViewer

#define IMG_HEIGHT 900
#define IMG_WIDTH 1023

- (id)initWithFrame:(NSRect)frame {
    self = [super initWithFrame:frame];
    if (self) {
        // Initialization code here.
            displayImage = [[NSBitmapImageRep alloc]
initWithBitmapDataPlanes:NULL

               pixelsWide:IMG_WIDTH

               pixelsHigh:IMG_HEIGHT

            bitsPerSample:8

          samplesPerPixel:4
```

```
                    hasAlpha:YES

                    isPlanar:NO

          colorSpaceName:NSCalibratedRGBColorSpace

               bytesPerRow:4*IMG_WIDTH

              bitsPerPixel:32];
             [self setNeedsDisplay:YES];
    }
    return self;
}

- (void)drawRect:(NSRect)dirtyRect {
    // Drawing code here.
      [displayImage draw];
}

- (void)SetDisplayImage:(NSBitmapImageRep *)image {
      if(displayImage){
            [displayImage dealloc];
      }
      displayImage = image;

      [self setNeedsDisplay:YES];
}

@end

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////



////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//
//  EchoGeneratorAppDelegate.m
//  EchoGenerator
//
//

#import "EchoGeneratorAppDelegate.h"
#include "loadimage.h"
#include "logging.h"


@implementation EchoGeneratorAppDelegate

@synthesize window;

- (void)applicationDidFinishLaunching:(NSNotification *)aNotification {
      // Insert code here to initialize your application
      [self EchoSetupChange: self];
```

```
      imageToEchoSetup.height = 2880;
      imageToEchoSetup.width  = 2880;
      imageToEchoSetup.carrierFreq = 4.5e9f;
      imageToEchoSetup.rangeScale = 10.0;

      antennaPattern = NULL;
      imageData = NULL;
      rangeData = NULL;
      quadDemodSignal = NULL;
      echo = NULL;

      imgexposure = 1.0;

}

- (IBAction) EchoSetupChange: (id)Sender{
      float minRangeVal = [minRange floatValue];
      float maxRangeVal = [maxRange floatValue];
      int echoBinCalc;
      char foo[32];
      // ensure that maxRange is always greater than min range to avoid the
      // need for further error checking
      if (minRangeVal > maxRangeVal) {
            maxRangeVal = minRangeVal + 0.0000001;
            sprintf(foo, "%f", maxRangeVal);
            [maxRange setStringValue: [NSString stringWithUTF8String: foo]];
//          LogInfo(@"minRange Greater Than maxRange, setting maxRange to
minRange + 0.0000001");
      }
      imageToEchoSetup.minRange = minRangeVal;
      imageToEchoSetup.maxRange = maxRangeVal;
      imageToEchoSetup.basebandBW = [basebandBW floatValue];
      imageToEchoSetup.pulseDuration = [pulseDur floatValue];
      echoBinCalc = ceil(((2.0 * (maxRangeVal - minRangeVal) / SPEED_OF_LIGHT)
+
                                    imageToEchoSetup.pulseDuration) * (2.0 *
imageToEchoSetup.basebandBW));
      // Not sure why, but the effect of some of the matlab code is to round
this figure to an
      // even value.  To match this behaviour, the following line is added.
      echoBinCalc += echoBinCalc & 0x1;
      imageToEchoSetup.numEchoBins = echoBinCalc;
      imageToEchoSetup.numQuadDemodSamples =
ceil(imageToEchoSetup.pulseDuration * 2.0 *

      imageToEchoSetup.basebandBW) + 1;

      sprintf(foo, "%i", echoBinCalc);
      [echoBins setStringValue:[NSString stringWithUTF8String: foo]];
      sprintf(foo, "%i", imageToEchoSetup.numQuadDemodSamples);
      [demodSamples  setStringValue:[NSString stringWithUTF8String: foo]];

}

- (IBAction) LoadAntennaPattern: (id)Sender{
```

```
//    LogInfo(@"Entered");
      NSOpenPanel *panel = [NSOpenPanel openPanel];
      [panel setAllowedFileTypes:[NSArray arrayWithObject:@"exr"]];
      [panel setAllowsOtherFileTypes:YES];
      [panel setAllowsMultipleSelection:NO];
      [panel setTitle:[NSString stringWithUTF8String:"Select Antenna Pattern
File"]];
      NSInteger panReturn = [panel runModal];

      if (panReturn == NSOKButton){
//          LogInfo(@"OK Button");
      } else if (panReturn == NSCancelButton) {
//          LogInfo(@"Cancel Button");
            return;
      } else {
//          LogInfo(@"Unrecognized return = %3d", panReturn);
            return;
      }

      NSURL *openFile = [panel URL];
//    LogInfo(@"filename = %@", openFile);

      int imgHeight, imgWidth;
      int err = GetImageSize([[openFile path] UTF8String], &imgWidth,
&imgHeight);

      if (err < 0) {
//          LogError(@"Error return from GetImageSize = %d", err);
            return;
      }

      if (antennaPattern != NULL){
            delete[] antennaPattern;
            antennaPattern = NULL;
      }

      antennaPattern = new float[imgWidth * imgHeight];

      if (rangeMod != NULL){
            delete[] rangeMod;
            rangeMod = NULL;
      }

      rangeMod = new float[imgWidth * imgHeight];

      [height setIntegerValue: imgHeight];
      [width setIntegerValue: imgWidth];
      imageToEchoSetup.height = imgHeight;
      imageToEchoSetup.width = imgWidth;

      float carrier;
      float patternAngle;
      float antLength;
      float rScaleVar;
```

```
      err = LoadAntennaPattern([[openFile path] UTF8String], imgWidth,
imgHeight,
                                               antennaPattern, rangeMod, &carrier,
&patternAngle, &antLength, &rScaleVar);

      if (err < 0) {
//          LogError(@"Error return from LoadAntennaPattern = %d", err);
            return;
      }

      imageToEchoSetup.carrierFreq = carrier;
      imageToEchoSetup.antLength = antLength;
      imageToEchoSetup.antPatAngle = patternAngle;
      imageToEchoSetup.rangeScale = rScaleVar;

      [carrierFreq setFloatValue:carrier];
      [rangeScale setFloatValue:rScaleVar];
}

NSInteger MyCompareUrl(NSURL *num1, NSURL *num2, void *context) {
      NSString *num1Path = [num1 path];
      NSString *num2Path = [num2 path];

      return [num1Path caseInsensitiveCompare: num2Path];
}

- (NSInteger) DisplayEcho:(std::complex<float> *)echoData height:(int)
imgheight width:(int) imgwidth{
      NSBitmapImageRep * image = [[NSBitmapImageRep alloc]
initWithBitmapDataPlanes:NULL

                           pixelsWide:imgwidth

                           pixelsHigh:imgheight

                        bitsPerSample:8

                      samplesPerPixel:4

                                hasAlpha:YES

                                isPlanar:NO

                       colorSpaceName:NSCalibratedRGBColorSpace

                            bitmapFormat:0

                             bytesPerRow:4 * imgwidth

                            bitsPerPixel:32];
      int i, j;
      std::complex<float> pixel;
      unsigned char *imgdata = [image bitmapData];
      float real, imaginary;
```

```
      for (j=0; j<imgheight; j++) {
            for (i=0; i<imgwidth; i++) {
                  int drawIndex = j * imgwidth + i;

                  pixel = echoData[drawIndex];
                  real = pixel.real();
                  imaginary = pixel.imag();

                  real *= imgexposure;
                  imaginary *= imgexposure;
                  if(real < 0.0) real = 0.0;
                  if(imaginary < 0.0) imaginary = 0.0;
                  if(real > 1.0) real = 1.0;
                  if(imaginary > 1.0) imaginary = 1.0;

                  imgdata[drawIndex*4] = real*255;
                  imgdata[drawIndex*4+1] = imaginary*255;
//                imgdata[drawIndex*4+1] = 0;
                  imgdata[drawIndex*4+2] = 0;
                  imgdata[drawIndex*4+3] = 255;
            }
      }

      [echoImage SetDisplayImage:image];

      return 0;
}

- (void) ExposureChange: (id)Sender {
      imgexposure = pow(2,[ExposureSetting floatValue]);

      [self DisplayEcho:echo height: numFiles width:
imageToEchoSetup.numEchoBins];
}

- (IBAction) LoadRangeSamples: (id)Sender{

      if (antennaPattern == NULL) {
//          LogError(@"Antenna Pattern Data not loaded!");
            return;
      }

      int imgHeight = imageToEchoSetup.height;
      int imgWidth = imageToEchoSetup.width;

      if (imageData != NULL) {
            delete[] imageData;
            imageData = NULL;
      }
      if (rangeData != NULL) {
            delete[] rangeData;
            rangeData = NULL;
      }
      imageData = new float[imgWidth * (imgHeight+1)];
      rangeData = new float[imgWidth * (imgHeight+1)];
```

```
      if (quadDemodSignal != NULL) {
            delete[] quadDemodSignal;
            quadDemodSignal = NULL;
      }
      quadDemodSignal = new
std::complex<float>[imageToEchoSetup.numQuadDemodSamples];

      int err = QuadDemodXmitSignalCalculate(&imageToEchoSetup,
quadDemodSignal);

      NSOpenPanel *panel = [NSOpenPanel openPanel];
      [panel setAllowedFileTypes:[NSArray arrayWithObject:@"exr"]];
      [panel setAllowsOtherFileTypes:YES];
      [panel setAllowsMultipleSelection:YES];
      [panel setTitle:[NSString stringWithUTF8String:"Select Platform Track
Images"]];
      NSInteger panelReturn = [panel runModal];

      if (panelReturn == NSOKButton) {
//          LogInfo(@"OK Button");
      } else if (panelReturn == NSCancelButton) {
//          LogInfo(@"Cancel Button");
            return;
      } else {
//          LogInfo(@"Unrecognized Return = %3d", panelReturn);
            return;
      }

//    NSArray *alphabeticalFileList = [panel URLs];
      NSArray *alphabeticalFileList = [[panel URLs]
sortedArrayUsingFunction:MyCompareUrl context:NULL];

      numFiles = [alphabeticalFileList count];

      if (echo != NULL) {
            delete[] echo;
            echo = NULL;
      }
      int numBins = imageToEchoSetup.numEchoBins;
      echo = new std::complex<float>[numFiles * numBins];

      for (int i = 0; i < numFiles; i++) {
            NSString *filePath = [[alphabeticalFileList objectAtIndex:i] path];
            //          LogInfo(@"Processing File: %@", filePath);
            err = LoadImageData([filePath UTF8String],
                                      imgWidth, imgHeight, imageData,
rangeData);
            if (err < 0) {
                  //                LogError(@"LoadImageData returned an error
= %d", err);
                  return;
            }
            err = ImageToEcho(&imageToEchoSetup, antennaPattern, rangeMod,
imageData, rangeData,
```

```
                                        quadDemodSignal, &echo[i * numBins]);
            if (err < 0) {
//                LogError(@"ImageToEcho returned an error = %d", err);
                  return;
            }

      }
//    LogInfo(@"Echo Generation complete! Displaying results.");
//    numFiles++;
      [self DisplayEcho:echo height: numFiles width: numBins];
}

- (IBAction) SaveEchoData: (id)Sender {
      if(echo == NULL)
            return;

      float settings[SETTING_TOTAL_NUMBER];

      settings[SETTING_MIN_RANGE_OFFSET] = imageToEchoSetup.minRange;
      settings[SETTING_MAX_RANGE_OFFSET] = imageToEchoSetup.maxRange;
      settings[SETTING_CARRIER_FREQ_OFFSET] = imageToEchoSetup.carrierFreq;
      settings[SETTING_BASEBAND_BW_OFFSET] = imageToEchoSetup.basebandBW;
      settings[SETTING_PULSE_DURATION_OFFSET] = imageToEchoSetup.pulseDuration;
      settings[SETTING_RANGE_SCALE_OFFSET] = imageToEchoSetup.rangeScale;
      settings[SETTING_ANTENNA_LENGTH_OFFSET] = imageToEchoSetup.antLength;
      settings[SETTING_ANTENNA_PATTERN_ANGLE_OFFSET] =
imageToEchoSetup.antPatAngle;

      NSLog(@"doSaveAs");
      NSSavePanel *tvarNSSavePanelObj = [NSSavePanel savePanel];
      int tvarInt = [tvarNSSavePanelObj runModal];
      if(tvarInt == NSOKButton){
            NSLog(@"doSaveAs we have an OK button");
      } else if(tvarInt == NSCancelButton) {
            NSLog(@"doSaveAs we have a Cancel button");
            return;
      } else {
            NSLog(@"doSaveAs tvarInt not equal 1 or zero = %3d",tvarInt);
            return;
      } // end if
      NSString * tvarDirectory = [tvarNSSavePanelObj directory];
      NSLog(@"doSaveAs directory = %@",tvarDirectory);
      NSString * tvarFilename = [tvarNSSavePanelObj filename];
      NSLog(@"doSaveAs filename = %@",tvarFilename);

      SaveEchoData([tvarFilename UTF8String], imageToEchoSetup.numEchoBins,
numFiles, echo, settings);

}

@end

////////////////////////////////////////////////////////////////////////////////////////////////////
```

3. RDA MATLAB CODE

```
////////////////////////////////////////////////////////////////////////////////////////////////////

function ExrEchoToRDA()
clear

% INITIALIZATION

% Load the echo data from the EXR file
filename = uigetfile;
[echoData, settings] = exr_rd_echo(filename);

numEchos = size(echoData, 1); % Number of pulse (Azimuth) samples
numEchos = 900;
rbins=size(echoData, 2); % Number of time (Range) samples

% Main SAR Parameters
PRF=300; % Pulse Repetition Frequency (Hz)
dur=numEchos / PRF; % Time of Flight (sec), PRF*dur = received echoes
vp=200; % Velocity of platform
fo=settings(3); % Carrier frequency
La=settings(7); % Antenna length actual
Xmin=settings(1); % Minimum Range
Xmax=settings(2); % Maximum Range
Xc = (Xmax + Xmin)/2;   % Range distance to center of target area
Tp=settings(5); % Chirp Pulse Duration
B0=settings(4); % Baseband bandwidth is plus/minus B0

% General Variables
cj=sqrt(-1);
c=3e8; % Propagation speed
ic=1/c; % Propagation frequency
lambda=c/fo; % Wavelength (6cm for fo = 4.5e9)
eta=linspace(0,dur,numEchos)'; % Slow Time Array

% Range Parameters
Kr=B0/Tp; % Range Chirp Rate
dt=1/(2*B0); % Time Domain Sampling Interval
Ts=(2*(Xmin))/c; % Start time of sampling
Tf=(2*(Xmax))/c+Tp; % End time of sampling

% Azimuth Parameters
Ka=(2*vp^2)./(lambda*Xc); % Linear Azimuth FM rate

% Measurement Parameters
t=Ts+(0:rbins-1)*dt; % Time array for data acquisition
s=echoData; % Echoed signal array


% RANGE DOPLER ALGORITHM (RDA)
% Range Reference Signal
td0=t-2*(Xc/c);
pha20=pi*Kr*((td0.^2)-td0*Tp);
```

```
s0=exp(cj*pha20).*(td0 >= 0 & td0 <= Tp);
fs0=fty(s0); % Reference Signal in frequency domain

% Power equalization
amp_max=1/sqrt(2); % Maximum amplitude for equalization
afsb0=abs(fs0);
P_max=max(afsb0);
I=find(afsb0 >= amp_max*P_max);
fs0(I)=((amp_max*(P_max^2)*ones(1,length(I)))./afsb0(I))...
    .*exp(cj*angle(fs0(I)));
deltaR=(lambda^2*(Xc).*(Ka*(dur*0.5-eta)).^2)/(8*vp^2); % RCM
cells=round(deltaR/.56); % .56 meters/cell in range direction
fs=zeros(numEchos,rbins); fsm=fs; fsmb=fs; smb=fs; fsac=fs; sac=fs;

% Range Compression
for k=1:(numEchos);
    fs(k,:)=fty(s(k,:));
    fsm(k,:)=fs(k,:).*conj(fs0);
    smb(k,:)=ifty(fsm(k,:));
end;

% Plot Range Compression Results
figure(2), imagesc(abs(smb))
xlabel('Range, samples'), ylabel('Azimuth, samples')


% Azimuth Reference Signal
smb0=exp(cj*pi*Ka.*eta.*(2*eta(numEchos/2+1)-eta));
fsmb0=ftx(smb0); % Azimuth Matched Filter Spectrum
for l=1:rbins;
    fsmb(:,l)=ftx(smb(:,l)); % Azimuth Fourier Transform
end;

% Range Cell Migration Correction (RCMC)
for k=1:numEchos/2;
    for m=1:rbins-9
        fsmb(k,m)=fsmb(k,m+cells(k));
        fsmb(numEchos-k,m)=fsmb(numEchos-k,m+cells(k));
    end
end;

% Azimuth Compression
for l=1:rbins;
    fsac(:,l)=fsmb(:,l).*conj(fsmb0); % Azimuth Matched Filtering
    sac(:,l)=iftx(fsac(:,l)); % Final Target Image
end;

% Plot Final Results
figure(1), imagesc(abs(sac))
xlabel('Range, samples'), ylabel('Azimuth, samples')

////////////////////////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////////////////////////
```

```
/*
 * MATLAB MEX function for reading echo data from EXR file.
 *
 */

#include "ImathBox.h"
#include "ImfInputFile.h"
#include "ImfArray.h"
#include "ImfChannelList.h"
#include "ImfPixelType.h"
#include "ImfStandardAttributes.h"
#include "Iex.h"

#include "mex.h"

using namespace Imf;
using namespace Imath;
using namespace Iex;

#define SETTING_MIN_RANGE_OFFSET                        0
#define SETTING_MAX_RANGE_OFFSET                        1
#define SETTING_CARRIER_FREQ_OFFSET                             2
#define SETTING_BASEBAND_BW_OFFSET                      3
#define SETTING_PULSE_DURATION_OFFSET                   4
#define SETTING_RANGE_SCALE_OFFSET                      5
#define SETTING_ANTENNA_LENGTH_OFFSET                   6
#define SETTING_ANTENNA_PATTERN_ANGLE_OFFSET    7

#define SETTING_TOTAL_NUMBER                                    8



/* Check inputs
 * one input: string (row vector of chars)
 *
 * 3 outputs:
 *     float array of real echo portions
 *     float array of imaginary echo portions
 *     float array of settings
 *
 * mexErrMsgTxt returns control directly to Matlab without executing any more of
 * the C code.  It is not necessary to perform an error return, and check data.
 */
void checkInputs(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[]) {

       if (nrhs != 1)
              mexErrMsgTxt("Incorrect number of input arguments.");

       if (nlhs != 2)
              mexErrMsgTxt("Incorrect number of output arguments.");

       if (mxIsChar(prhs[0]) != 1)
              mexErrMsgTxt("Input must be a string.");

       if (mxGetM(prhs[0]) != 1)
              mexErrMsgTxt("Input must be a row vector.");
```

```
	return;
}

/* Read the settings attributes from the file
 *
 * function is called during a try/catch block.  throwing the errors
 * will invoke the catch routine, and exit the function gracefully.
 */
void ReadAttributes(InputFile &file, double *settingsArray){

	const FloatAttribute *minRange = file.header().findTypedAttribute
                                                        <FloatAttribute> ("min
range");
	if (minRange == NULL){
		throw "Min Range attribute not found";
	}
	settingsArray[SETTING_MIN_RANGE_OFFSET] = minRange->value();

	const FloatAttribute *maxRange = file.header().findTypedAttribute
                                                        <FloatAttribute> ("max
range");
	if (maxRange == NULL){
		throw "Max Range attribute not found";
	}
	settingsArray[SETTING_MAX_RANGE_OFFSET] = maxRange->value();

	const FloatAttribute *frequency = file.header().findTypedAttribute
                                                        <FloatAttribute>
("frequency");
	if (frequency == NULL){
		throw "Frequency attribute not found";
	}
	settingsArray[SETTING_CARRIER_FREQ_OFFSET] = frequency->value();

	const FloatAttribute *baseBw = file.header().findTypedAttribute
                                                        <FloatAttribute> ("baseband
bw");
	if (baseBw == NULL){
		throw "Baseband Bandwidth attribute not found";
	}
	settingsArray[SETTING_BASEBAND_BW_OFFSET] = baseBw->value();

	const FloatAttribute *pulseDur = file.header().findTypedAttribute
                                                        <FloatAttribute> ("pulse
duration");
	if (pulseDur == NULL){
		throw "Pulse Duration attribute not found";
	}
	settingsArray[SETTING_PULSE_DURATION_OFFSET] = pulseDur->value();

	const FloatAttribute *rangeScale = file.header().findTypedAttribute
                                                        <FloatAttribute> ("range
scale");
	if (rangeScale == NULL){
```

```
                throw "Range Scale attribute not found";
        }
        settingsArray[SETTING_RANGE_SCALE_OFFSET] = rangeScale->value();

        const FloatAttribute *antLen = file.header().findTypedAttribute
                                                                <FloatAttribute> ("antenna
length");
        if (antLen == NULL){
                throw "Antenna Length attribute not found";
        }
        settingsArray[SETTING_ANTENNA_LENGTH_OFFSET] = antLen->value();

        const FloatAttribute *antAngle = file.header().findTypedAttribute
                                                                <FloatAttribute> ("antenna
angle");
        if (antAngle == NULL){
                throw "Antenna Angle attribute not found";
        }
        settingsArray[SETTING_ANTENNA_PATTERN_ANGLE_OFFSET] = antAngle->value();

}


/* Main function called from Matlab.
 *
 * First checks to make sure the function was called correctly,
 * then calls a function to load the settings that were saved in the attributes of the
files.
 * finally reads the data from the file (real in Red "R" channel, imaginary in Green "G"
 * channel, and copies it into the Matlab array.
 */
void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[]) {

        checkInputs(nlhs, plhs, nrhs, prhs);
        char *filename = mxArrayToString(prhs[0]);

        try {
                InputFile file(filename);
                mxFree(filename);

                // read the settings from the echo file.
                int dims[2] = {SETTING_TOTAL_NUMBER, 1};
                plhs[1] = mxCreateNumericArray(2, dims, mxDOUBLE_CLASS, mxREAL);
                double *settings = mxGetPr(plhs[1]);
                ReadAttributes(file, settings);

                // get the image dimensions
                Box2i dw = file.header().dataWindow();

                int x  = dw.max.x - dw.min.x + 1;
                int y = dw.max.y - dw.min.y + 1;

                // allocate the buffers for the floating point data from the file
                Array2D<float> re(y,x);
                Array2D<float> im(y,x);
```

```
		// add the buffers to the frame buffer construct to prepare for reading
		FrameBuffer frmBuf;
		frmBuf.insert("R", Slice(FLOAT, (char *)&re[0][0], sizeof(float),
sizeof(float) * x, 1, 1, 0.0));
		frmBuf.insert("G", Slice(FLOAT, (char *)&im[0][0], sizeof(float),
sizeof(float) * x, 1, 1, 0.0));

		// read the data from the file
		file.setFrameBuffer(frmBuf);
		file.readPixels(dw.min.y, dw.max.y);

		// allocate the Matlab variable space
		dims[0] = y;
		dims[1] = x;
		plhs[0] = mxCreateNumericArray(2, dims, mxDOUBLE_CLASS, mxCOMPLEX);
		double *echoReal = mxGetPr(plhs[0]);
		double *echoImaginary = mxGetPi(plhs[0]);

		// copy the data from the "C" constructs to the Matlab constructs.
		for (int i = 0; i < y; ++i) {
			for (int j = 0; j < x; ++j) {
				int k = j*y + i;

				echoReal[k]		= re[i][j];
				echoImaginary[k]	= im[i][j];
			}
		}

	// if anything goes wrong from the try { this next }, this
	// block of code will get called, clean up allocated memory, and
	// print an error message in the Matlab window.
	} catch (const std::exception &exc) {
		mxFree(filename);
		mexErrMsgTxt(exc.what());
	}


	return;
}


////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```